Parallelization of Stochastic Estimation Algorithms on Multicore Computational Platforms

Olov Rosén
Parallelization of Stochastic Estimation Algorithms on Multicore Computational Platforms

Olov Rosén
olov.rosen@it.uu.se

April 2013

Division of Systems and Control
Department of Information Technology
Uppsala University
Box 337
SE-751 05 Uppsala
Sweden

http://www.it.uu.se/

Dissertation for the degree of Licentiate of Philosophy in Electrical Engineering with Specialization in Automatic Control

© Olov Rosén 2013
ISSN 1404-5117
Printed by the Department of Information Technology, Uppsala University, Sweden
Abstract

The main part of this licentiate thesis concerns parallelization of recursive estimation methods, both linear and nonlinear. Recursive estimation deals with the problem of extracting information about parameters or states of a dynamical system, given noisy measurements of the system output and plays a central role in many applications of signal processing, system identification, and automatic control. Solving the recursive Bayesian estimation problem is known to be computationally expensive, which often makes the methods infeasible in real-time applications and problems of large dimension. As the computational power of the hardware is today increased by adding more processors on a single chip rather than increasing the clock frequency and shrinking the logic circuits, parallelization is the most powerful way of improving the execution time of an algorithm. It has been found in this thesis that several of the optimal filtering methods are suitable for parallel implementation, in certain ranges of problem sizes. It has been concluded from the experiments that substantial improvements can be achieved by performing "tailor"-made parallelization, compared to straightforward implementations based on multi-threaded libraries. For many of the suggested parallelizations, a linear speedup in the number of cores has been achieved that have provided up to 8 times speedup on a double quad-core computer. As the evolution of the parallel computer architectures is unfolding rapidly, many more processors on the same chip will become available. The developed methods do not, of course, scale infinitely, but definitely can exploit and harness some of the computational power of the next generation of parallel platforms, allowing for optimal state estimation in real-time applications.
Acknowledgment

I would like to thank my supervisor Professor Alexander Medvedev for his support, guidance, and for sharing his knowledge and expertise throughout this work. I would also like to thank all my colleagues at SysCon for providing such a pleasant working atmosphere. The thesis covers research within the project "Computationally Demanding Real-Time Applications on Multicore Platforms" funded by Swedish Foundation for Strategic Research, whose financial support is greatly appreciated. My deepest gratitude goes to Linnéa for her love and support and last but not least I would like to thank my family, for their support and encouragement.
Contents

1 Introduction 1

2 Contribution and publications 3

3 Nomenclature 5

4 Some probability theory 6
   4.1 Definition of a random variable 6
   4.2 The distribution function and some associated measures 6
   4.3 Estimation from random samples 7

5 Recursive Bayesian estimation 9
   5.1 Kalman filter 11
      5.1.1 General Kalman filter 11
      5.1.2 Static Kalman filter 12
      5.1.3 Extended Kalman filter 12
      5.1.4 Unscented Kalman filter 13
   5.2 Monte-Carlo methods 14
   5.3 Grid-based methods 14
   5.4 Computational complexity 16

6 Multicore architecture 17
   6.1 Evolution of the multicore processor 17
   6.2 Parallel architectures 17

7 Parallel implementation 19
   7.1 Parallel implementation 19
   7.2 Software 20
   7.3 Performance measures 21
   7.4 Data partitioning 22
   7.5 High-performance computing 24

8 Parallel implementation of filtering methods 26
   8.1 Kalman filter 26
      8.1.1 General Kalman filter 26
      8.1.2 Static Kalman filter 26
      8.1.3 Extended Kalman filter 28
      8.1.4 UKF 28
   8.2 Grid-based methods 29
   8.3 Particle filter 31
   8.4 Series expansion 31

9 Concluding remarks 32
1 Introduction

The main part of this licentiate thesis deals with parallelization of recursive estimation methods, both linear and nonlinear. Recursive estimation deals with the problem of extracting information about parameters or states of a dynamical system, given noisy measurements of the system output. Recursive estimation plays a central role in many applications of signal processing, system identification and automatic control. In general, there is no closed-form analytical solution to the problem. An exception is the case of a linear system with white Gaussian process and measurement noise, in which case the analytical solution is known as the Kalman filter. For other system classes, approximation methods have to be employed to solve the problem. An approximative solution can be made arbitrary accurate at the price of an increased computational burden. For applications that require high sampling rates and/or are of high dimension, the computational cost in many cases renders the solution infeasible. It has generally been found that the recursive Bayesian estimation problem is amenable to parallelization, and substantial improvements in the execution time can be achieved by execution on parallel hardware.

Today, the computational capacity is increased by adding more processors on a single chip, rather than shrinking the logic circuits and increasing the processor operating frequency. This is due to physical limitations as well as power and heat dissipation concerns. All major manufacturers have turned from a single core design to a multicore design and parallel processing is no longer the exclusive domain of supercomputers or clusters. Any computer bought today is likely to have two or more cores, and the number of cores available on a single chip increases steadily. To utilize the computational power provided by parallel hardware, algorithms must be implemented in a way that suits the parallel architecture. Occasionally, the implementation is rather straightforward but in many cases the algorithm must, at some point, be modified to yield better parallelization properties, where the modification often comes with a decreased accuracy, sacrificed in favor of faster execution time.

In real-time applications, it is common that suboptimal filtering methods are employed because the optimal solution simply requires too much computation to be practically feasible. For example, recursive least squares and least mean squares methods are commonly used for linear filtering, instead of the optimal Kalman filter which can provide both faster convergence rates and better mean square error. For nonlinear systems, the (suboptimal) extended Kalman filter is a usual choice even though other nonlinear methods such as the unscented Kalman filter, grid-based methods and simulation-based methods can provide superior estimation accuracy. With the computational power offered by parallel hardware, new doors are opening for the application of computationally costly optimal methods. As parameter and state estimation constitute a key part in automatic control, system identification and signal processing systems, estimation quality can be of utmost importance for the performance of the system. This motivates the interest for designing implementations of recursive
estimation methods that can be executed on a parallel architecture and provide real-time feasibility. Another aspect in the parallelization, important for e.g. mobile devices powered by batteries and low power communication systems, is the substantially lower power-per-FLOP ratio for parallel processors, compared to sequential processors.

The main part of this thesis deals with the parallelization of recursive estimation problems in discrete time. Another problem, by its Bayesian nature related to the estimation problem, is anomaly detection, to which a smaller part of the thesis is devoted. Anomaly detection refers to finding patterns in a given data set that do not conform to a properly defined normal behavior. A model-free and computationally light method has been developed and is presented in Paper V.

The thesis is divided into nine sections. Sec. 2 summarizes the contribution and the papers that are included in the thesis. In Sec. 3, the nomenclature is given. Some concepts in probability theory are briefly reviewed in Sec. 4. In Sec. 5, the recursive Bayesian estimation problem is stated and several methods for its solution are discussed. In Sec. 6, the multicore architecture is explained. Some general concepts and properties of parallel implementations are stated in Sec. 7. In Sec. 8, a parallel implementation of the recursive Bayesian estimation problem is discussed in detail. Finally, concluding remarks are provided in Sec. 9.
2 Contribution and publications

During the work on this licentiate thesis, several recursive estimation (filtering) methods have been studied with their parallelization on multicore architectures in mind. The parallelizations are presented in Sec. 8, with references to the corresponding papers. A number of papers has been published in conference proceedings and journals but some of the material is yet unpublished. The contributions can be summarized as follows:


Implementation and evaluation in terms of speedup and tracking accuracy of four different parallelizations of the particle filter for a bearings-only tracking problem have been performed. It has been found that a linear speedup in the number of cores can be achieved with a small loss in estimation accuracy. The paper also treats parallelization of the Kalman filter for parameter estimation. It has been found that the implementation can be made highly efficient and linear speedup in the number of processors used has been achieved.


A parallelization of the general formulation of the Kalman filter has been developed. For efficiency, it is required that the system is banded, which, as shown in the paper, is a rather non-restrictive assumption. The proposed parallelization is applied to a wideband code division multiple access application, a field where the computational cost is growing rapidly due to the increasing number of smart phones in the system. The parallelization is compared to an automatic BLAS implementation and is shown to be able to reach linear speedup, whereas the BLAS implementation exhibits a speedup no higher than two times.

**Paper III** O. Rosen and A. Medvedev *Nonlinear Filtering Using Orthogonal Basis Functions*, submitted to a conference.

A novel solution method to the recursive Bayesian estimation problem has been derived. The solution is based on representing the involved probability density functions (PDFs) in orthogonal bases. Owing to the orthogonality of the bases, the method is highly amenable to parallel implementation.

**Paper IV** O. Rosen and A. Medvedev *Non-Parametric Anomaly Detection in Trajectorial Data* submitted to a journal.

A novel method for anomaly detection for trajectory following systems has been developed. The method estimates the PDFs associated with
normal behavior from a set of system realizations and is hence model-free. The method has been applied to an eye-tracking application where the eye movements of a person are studied in order to detect deviating characteristics that can potentially indicate malfunction in the oculomotor system. The method has also been applied to a surveillance problem where deviating behaviors of vessels traveling through the English channel are sought to detect.

**Additional material** Parallelizations of the static Kalman filter, extended Kalman filter and the unscented Kalman filter have been performed. The parallelizations are based on the BLAS library, and some of the methods given in Papers I-III. The implementations have been evaluated in terms of execution time and scalability. Discussion and analysis are provided in Sec. 8.

A list of other publications related to but not included in the thesis is given below:


3 Nomenclature

Symbols

\textbf{A} Matrices are written in bold upper case letters.

\textbf{x} Vectors are written in bold lower case letters.

\textbf{X} Stochastic variable.

\(p(\mathbf{x})\) Probability density function for stochastic variable \(X\).

\(p_X(\mathbf{z})\) Probability density function for stochastic variable \(X\) evaluated at \(\mathbf{z}\).

\(p(\mathbf{x}, \mathbf{y})\) Joint density function for random variables \(X\) and \(Y\).

\(p(\mathbf{x}|\mathbf{y})\) Conditional density function for \(p(\mathbf{x}, \mathbf{y})\) given \(Y = \mathbf{y}\).

\(f(\mathbf{x})\) Vector-valued function.

\(\mathbb{R}^n\) \(n\)-dimensional set of real numbers.

\(\mathbb{N}^n\) \(n\)-dimensional set of natural numbers.

\(m : n\) Set of numbers \(\{m, m + 1, \ldots, n\}\), \(m, n \in \mathbb{N}\) and \(m \leq n\).

\(\mathcal{N}(\mu, \Sigma)\) Normal distribution with mean \(\mu\) and covariance \(\Sigma\).

\(\gamma(\mathbf{x}; \mu, \Sigma)\) Probability density function for normal distribution \(\mathcal{N}(\mu, \Sigma)\).

\(x_{m:n}\) The ordered set \(\{x_m, x_{m+1}, \ldots, x_n\}\).

\(Pr(A)\) Probability of a random event \(A\).

Abbreviations

KF Kalman filter.

EKF Extended Kalman filter.

UKF Unscented Kalman filter.

PF Particle filter.

PDF Probability density function.

CPU Central processing unit.

SMC Shared memory multicore.

MIMO Multiple input multiple output.

SISO Single input single output.

MISO Multiple input single output.

FLOP Floating point operation.

FLOPS Floating point operations per second.
4 Some probability theory

Probability theory constitutes a whole branch in mathematics and is one of the foundations that this thesis is built on. Here, some few concepts regarding random variables that are utilized in the thesis are summarized. Much of the material is assumed to be well known to the reader and is hence only briefly explained. The problem of estimating a probability density function from a random sample is discussed in more detail as it is a more specialized topic.

4.1 Definition of a random variable

A random variable is a mathematical object, developed to represent an event that has not yet happened and is subject to chance. A common example is the number of dots in a dice throw, which, if the dice is balanced, has a probability of 1/6 to be some of the numbers \{1, 2, 3, 4, 5, 6\}. To give a more formal definition of a random variable, the concept of a probability space has first to be introduced.

A probability space is defined as the triplet \((\Omega, A, P)\) where \(\Omega = \{\omega\}\) is a set of all possible outcomes. \(A = \{a\}\) is a set of events, where \(a \subseteq \Omega\) and \(P : A \to \mathbb{R}^+\) is a function that to each event \(a\) assigns a probability \(P(a) \geq 0\). A random variable, or stochastic variable, is defined as a real-valued function \(X : \Omega \to \mathbb{R}\) on the set \(\Omega\), [8].

4.2 The distribution function and some associated measures

Assume that \(X = [X_1, X_2, ..., X_n]^T\) is an \(n\)-dimensional random variable. To every stochastic variable, there is an associated distribution \(D\), which relationship is written as \(X \sim D\). To each \(D\), there are two commonly associated distribution functions, the cumulative density function (CDF), \(F_X(x)\) and the probability density function (PDF), \(f_X(x)\) defined as

\[
F_X(x) = Pr(X \leq x),
\]

\[
= Pr(X_1 \leq x_1, X_2 \leq x_2, ..., X_n \leq x_n),
\]

\[
f_X(x) = \frac{\partial^n F_X(x)}{\partial x_1 \partial x_2 ... \partial x_n}.
\]

When there is no risk for confusion, the index is usually dropped and the functions are written just as \(F(x)\) and \(f(x)\). Let \(Y\) be a subset of the random variables \(X_1, X_2, ..., X_n\) and \(Z\) be the subset that contains the variables not included in \(Y\). The conditional density function

\[
f(y|z) = \frac{f(y, z)}{f(z)}
\]
specifies the density of $Y$ given that $Z = z$. The marginal distribution, characterizing the distribution of $y$ alone, is calculated as

$$f(y) = \int f(y, z) dz.$$ 

The expected value of $g(X)$ where $g$ is an arbitrary function is given by

$$E[g(X)] = \int_{\mathbb{R}^n} g(x) f_X(x) dx.$$ 

The mean value and covariance of $X$ are defined as

$$\mu = E[X] = \int_{\mathbb{R}^n} x f_X(x) dx,$$

$$\Sigma = E[(X - \mu)(X - \mu)^T] = \int_{\mathbb{R}^n} (x - \mu)(x - \mu)^T f_X(x) dx.$$ 

### 4.3 Estimation from random samples

In many cases, the distribution of the random variable is unknown, but a set of observations from it is available. To be more precise: Let $\{X_i\}_{i=1}^N$ be a set of $N$ independent identically distributed random variables with distribution $D$. A sample from $D$ is given by the set of observations $\{x_i\}_{i=1}^N$ where $x_i$ is a realization of $X_i$. From the sample, information about the underlying distribution can be extracted. For instance, the sample mean and covariance

$$\hat{\mu} = \frac{1}{N} \sum_{i=1}^N x_i,$$

$$\hat{\Sigma} = \frac{1}{N-1} \sum_{i=1}^N (x_i - \hat{\mu})(x_i - \hat{\mu})^T,$$

are unbiased estimators of the true mean and variance of the distribution.

It can also be of interest to estimate the PDF of $D$ from the sample. This can be done in several ways: The histogram constitutes a piecewise constant estimator of the PDF. It is a simple but rather primitive estimator which requires a relatively large sample size $N$ to yield a good approximation. Two other commonly used approximation approaches are represented by kernel-based methods and orthogonal series estimators.

A kernel approximation, see e.g. [24], of $f(x)$ is given by

$$\hat{f}(x) = \frac{1}{N|H|^{1/2}} \sum_{i=1}^N \phi(H^{-1/2}(x - x_i)), $$

where $\phi(\cdot)$ is the kernel function that is symmetric and integrates to one. The parameter $H \in \mathbb{R}^{n \times n}$ is known as the bandwidth of the kernel, it is symmetric.
and positive definite and acts as a smoothing parameter. Assume that $H = hI$. A high value of $h$ will give a smooth estimate, with a low variance but a high bias. Conversely, a low value of $h$ will give a higher variance but a low bias of the estimate. Consider the one dimensional case. The value of $h$ is a user parameter, but there are some guidelines how it should be chosen. It can be shown [21] that the optimal choice for $h$, in the sense that it minimizes the asymptotic mean integrated square error, is given by

$$h = \hat{\sigma} C(\nu) N^{-\frac{1}{2\nu+1}}$$

where $\hat{\sigma}$ is the sample standard deviation, and $C$ and $\nu$ are kernel-specific constants. Under weak assumptions, it has been shown that the kernel estimator is optimal in the sense that there can be no non-parametric estimator that converges faster to the true density [28].

The kernel density estimator has the drawback that the approximation requires a large number of terms, namely $N$ of them, to be specific. An alternative to the kernel estimator is the orthogonal series estimator [23], [21] that has the capability of capturing the shape of $f(x)$ using far fewer terms than the kernel estimator. Using an orthogonal series estimator, the estimate is given by

$$\hat{f}(x) = \frac{1}{M} \sum_{i=1}^{M} a_i \varphi_i(x),$$

where $\{\varphi_i\}$ is a set of basis functions, orthogonal on the domain $\Psi$ w.r.t. the weighted inner product

$$< \varphi_i, \varphi_j > = \int_{\Psi} w(x)^2 \varphi_i(x) \varphi_j(x) dx.$$

If the function $f$ were known, the coefficients would be computed as

$$a_i = \int_{\mathbb{R}^n} w(x)f(x)\phi_i(x)dx = E[w(X)\phi_i(X)].$$

The coefficients can thus be unbiasedly estimated from the sample according to

$$a_i = E[w(X)\phi_i(X)] \approx \frac{1}{N} \sum_{j=1}^{N} w(x_j)\phi_i(x_j).$$

For the orthogonal series estimator, the number of terms $M$ in the expansion can somewhat loosely be interpreted as a smoothing parameter. A low value of $M$ will give a low variance of the estimate but a large bias, and vice versa for a high value of $M$. 

8
5 Recursive Bayesian estimation

Statistical estimation deals with the problem of estimating parameters, based on empirical data containing some random component. In recursive statistical estimation, the estimate is updated in an iterative manner as new evidence about the unobserved quantity is acquired. Being the underlying problem to all optimal stochastic filtering methods, the recursive Bayesian estimation problem is briefly reviewed in this section. A thorough exposition of the subject can be found in some of the classical textbooks, e.g. [2], [19], [27], [22].

Consider a stochastic process \( \{x_k, k = 0, 1, \ldots\} \) described by the state space model

\[
\begin{align*}
  x_{k+1} &= f_k(x_k) + v_k, \\
  y_k &= h_k(x_k) + e_k,
\end{align*}
\]

with the state \( x_k \in \mathbb{R}^n \) and output \( y_k \in \mathbb{R}^p \). The sequences \( v_k \) and \( e_k \) are zero-mean white noise processes characterized by the distributions with known PDFs given by \( p(v_k) \) and \( p(e_k) \), respectively, where \( k \) is the time step. The functions \( f_k(\cdot) \) and \( h_k(\cdot) \) are arbitrary but known vector-valued functions. A more general formulation includes as well an input signal. As the input is a deterministic quantity and straightforward to incorporate, it is omitted here for brevity.

In recursive Bayesian estimation the aim is, given the observed measurements \( y_{1:k} \), to make a statistical inference about the unobserved state \( x_k \). This is solved by recursively constructing the PDF \( p(x_k|y_{1:k}) \) at each step \( k \). From \( p(x_k|y_{1:k}) \), a state estimate can be extracted. It can be shown that the conditional mean

\[
\hat{x}_k = E[p(x_k|y_{1:k})]
\]

is the optimal estimator in the sense that it minimizes the variance of the estimation error [22]. However, sometimes it can be motivated to employ another estimator, especially in the case of a multimodal density function, such as e.g.

\[
\hat{x}_k = \arg \max_{x_k} p(x_k|y_{1:k}).
\]

In the recursive Bayesian estimation framework, \( p(x_k|y_{1:k}) \) is calculated by iterating the prediction and update steps. Assume that \( p(x_{k-1}|y_{1:k-1}) \) is known and that a new measurement \( y_k \) is obtained. The predicted PDF \( p(x_k|y_{1:k-1}) \) is obtained from the Kolmogorov-Chapman equation

\[
p(x_k|y_{1:k-1}) = \int_{\mathbb{R}^n} p(x_k|x_{k-1})p(x_{k-1}|y_{1:k-1})dx_{k-1}.
\]

From Bayes rule, the updated PDF \( p(x_k|y_{1:k}) \) is found, using the new evidence \( y_k \), from

\[
p(x_k|y_{1:k}) = \frac{p(y_k|x_k)p(x_k|y_{1:k-1})}{p(y_k|y_{1:k-1})}.
\]
Assume that measurements are available from time step $k = 1$. Given an initial PDF $p(x_0|y_0) = p(x_0)$ for the initial state $x_0$, the PDF $p(x_k|y_{1:k})$ is obtained by applying the prediction and update steps (3), (4) recursively to the measurements $y_{1:k}$. Fig. 1 shows an example of how the PDF evolves over time.

The PDFs $p(x_k|x_{k-1})$ and $p(y_k|x_{k-1})$ defined by (3), (4), are implicitly given by the state space model (1), (2). Via the generalized convolution formula, the expression for $p(x_k|x_{k-1})$ is found as

$$p(x_k|x_{k-1}) = \int_{\mathbb{R}^n} p(x_k|x_{k-1}, v_{k-1})p(v_{k-1})dv_{k-1}$$

$$= \int_{\mathbb{R}^n} \delta(x_k - f_{k-1}(x_{k-1}) - v_{k-1})p(v_{k-1})dv_{k-1}$$

$$= p_{v_{k-1}}(x_k - f_{k-1}(x_{k-1})),$$

and, similarly, $p(y_k|x_k)$ is given by

$$p(y_k|x_k) = p_{e_k}(y_k - h_k(x_k)).$$

In general, closed-form expressions cannot be obtained for (3)-(4). A special case arises when $f_k$ and $h_k$ are linear and $v_k$ and $e_k$ are Gaussian with zero mean, for which case the solution is given by the Kalman filter. For non-Gaussian noise, it can be shown that the Kalman filter is still the best unbiased linear estimator. Under other premises, the problem must be solved by approximation methods of which Monte Carlo methods and grid-based are most commonly used. The Kalman filter describes the PDF by a Gaussian
Figure 2: Representation of a 2 dimensional PDF (a) Kalman filter, (b) particle filter, (c) grid-based filter.

function, the Monte Carlo methods provide a sample from the distribution, and the grid-based methods approximate the PDF over a discrete set of grid points. Fig. 2 illustrates how the above methods represent the information about the sought PDF, illustrated by a 2-dimensional example.

In the following subsections, the core of each solution method is outlined. A plethora of minor modifications is available. However, the parallelization properties that are in the focal point of this thesis are in most cases not affected by the modifications.

5.1 Kalman filter

5.1.1 General Kalman filter

If the process and measurement noises are white Gaussian noise sequences and \( f_k, h_k \) are linear functions, (1)-(2) can be written as

\[
\begin{align*}
    x_{k+1} &= F_k x_k + v_k, \\
    y_k &= H_k x_k + e_k,
\end{align*}
\]

where \( F_k \in \mathbb{R}^{n \times n} \) and \( H_k \in \mathbb{R}^{p \times n} \). Under these assumptions, it can be shown that the prior and posterior distributions are Gaussian and hence completely characterized by the mean \( \mu \) and covariance \( \Sigma \). The closed-form solution of (3), (4) that propagates the mean \( \mu \) (coinciding with the estimated state \( \hat{x} \)) and the estimation error covariance

\[
P_{k|k} = E((x_k - \hat{x}_{k|k})(x_k - \hat{x}_{k|k})^T),
\]

is the Kalman filter [14], for which the prediction and update steps can be formulated as follows:

**Prediction**

\[
\begin{align*}
    \hat{x}_{k|k-1} &= F_{k-1} \hat{x}_{k-1|k-1}, \\
    P_{k|k-1} &= F_{k-1} P_{k-1|k-1} F_{k-1}^T + Q_{k-1},
\end{align*}
\]
Update

\[
K_k = P_{k|k-1}H_k^T(H_kP_{k|k-1}H_k^T + R_k)^{-1}, \quad (9)
\]

\[
\hat{x}_{k|k} = \hat{x}_{k|k-1} + K_k(y_k - H_k\hat{x}_{k|k-1}), \quad (10)
\]

\[
P_{k|k} = (I - K_kH_k)P_{k|k-1}, \quad (11)
\]

where \( Q_k = E[v_kv_k^T] \) and \( R_k = E[e_ke_k^T] \). There is a huge literature devoted to the Kalman filter, with some references given by [13], [22], [3], [11].

5.1.2 Static Kalman filter

Assume that the system (5)-(6) is time invariant. Under the mild assumptions of \( R \) being positive definite and \((F, B)\) being a stabilizable pair, where \( Q = BB^T \), the gain \( K_k \) and the error covariance \( P_{k|k} \) will converge to the constants \( K \) and \( P \) respectively, as \( k \to \infty \) [22]. The filter gain can then be treated as a static one, with \( P \) given by the solution to the algebraic Riccatti equation

\[
P = FP^TP + Q + FPH^T(HPH^T + R)^{-1}HP^TP,
\]

and \( K \) calculated as

\[
K = FPH(HPH^T + R)^{-1}.
\]

The static Kalman filter is then

\[
\hat{x}_{k+1|k} = F\hat{x}_{k|k-1} + K(y_k - H\hat{x}_{k|k-1}). \quad (12)
\]

5.1.3 Extended Kalman filter

A generalization of the KF that applies to nonlinear systems is the extended Kalman filter (EKF) [22], [11]. It is a suboptimal solution to the recursive estimation problem based on a linearization of the measurement and system equations around estimates of the state.

Assume that the nonlinear functions \( f_k \) and \( h_k \) are differentiable. At each time step they are approximated by a first order Taylor expansion, i.e.

\[
f_k(x_k) \approx f_k(\hat{x}_{k|k}) + F_k(x_k - \hat{x}_{k|k}),
\]

\[
h_k(x_k) \approx h_k(\hat{x}_{k|k-1}) + H_k(x_k - \hat{x}_{k|k-1}),
\]

where

\[
F_k = \left. \frac{\partial f_k(x)}{\partial x} \right|_{x = \hat{x}_{k|k-1}}, \quad (13)
\]

\[
H_k = \left. \frac{\partial h_k(x)}{\partial x} \right|_{x = \hat{x}_{k|k-1}}. \quad (14)
\]
The filtering is then performed by applying the standard KF equations (7)-(11) to the linearized system. If the nonlinearities are severe, the linearization can be a poor approximation of the system, which can in the worst case lead to divergence of the filter.

5.1.4 Unscented Kalman filter

Another filtering method that applies to nonlinear system and can be shown to be more robust against nonlinearities is the unscented Kalman filter (UKF) [12]. In the most basic setting, one iteration of the method can be summarized as follows.

Assume that \( x_{k-1} \) has the mean and covariance given by \( \mu_{k-1} \) and \( \Sigma_{k-1} \), respectively. The UKF propagates the first and second order moments, i.e. \( \mu \) and \( \Sigma \), of the distribution. A set of sigma points \( S_{k-1} = \{ x_{k-1}^{(i)}, w_{k-1}^{(i)} \}_{i=1}^N \) is chosen such that the sample mean and covariance of \( S_{k-1} \) are equal to \( \mu_{k-1} \) and \( \Sigma_{k-1} \), respectively, and \( \sum_{i=1}^N w_{k-1}^{(i)} = 1 \). A symmetric set of \( 2n \) points that satisfies mean \( (S_{k-1}) = \mu_{k-1} \) and cov \( (S_{k-1}) = \Sigma_{k-1} \) can be calculated for \( i = 1, 2, \ldots, n \) as

\[
\begin{align*}
x_{k-1}^{(i)} &= \mu_{k-1} + (\sqrt{n} \Sigma_{k-1})_i, \\
w_{k-1}^{(i)} &= n/2, \\
x_{k-1}^{(i+n)} &= \mu_{k-1} - (\sqrt{n} \Sigma_{k-1})_i, \\
w_{k-1}^{(i+n)} &= n/2,
\end{align*}
\]

where \( (\sqrt{n} \Sigma_{k-1})_i \) denotes the \( i \)-th row of the Cholesky factorization of \( n \Sigma_{k-1} \). The sigma points are then propagated through the nonlinear system equation, i.e.

\[
x_k^{(i)} = f_{k-1}(x_{k-1}^{(i)})
\]

From the updated set \( S_k = \{ x_k^{(i)}, w_k^{(i)} \}_{i=1}^N \) the estimated mean and covariance are calculated as

\[
\begin{align*}
\mu_k &= \sum_{i=1}^N w_k^{(i)} x_k^{(i)}, \\
\Sigma_k &= \sum_{i=1}^N w_k^{(i)} (x_k^{(i)} - \mu_k)(x_k^{(i)} - \mu_k)^T.
\end{align*}
\]
5.2 Monte-Carlo methods

A simple and powerful, though computationally costly, method to perform filtering via (3), (4) is by means of Monte-Carlo simulation [20], [6]. The Monte-Carlo based framework can handle nonlinear systems with general noise distributions. The method provides a sample $S_k = \{x_k^{(i)}, w_k^{(i)}\}_{i=1}^N$ from the distribution with PDF $p(x_k|y_{1:k})$ from which the desired information about the random variable $x_k$ can be extracted. Assume that at time step $k - 1$, $S_{k-1} = \{x_{k-1}^{(i)}, w_{k-1}^{(i)}\}_{i=1}^N$ constitutes a weighted sample from the distribution with PDF $p(x_{k-1}|y_{1:k-1})$, where $x_{k-1}^{(i)}$ is the $i$-th observation, called a particle, with associated weight $w_{k-1}^{(i)} \geq 0$. Given $S_{k-1}$, a sample from $p(x_k|y_{1:k-1})$ is obtained by propagating each particle through system equation (1), i.e.

$$x_k^{(i)} = f_{k-1}(x_{k-1}^{(i)}) + v_{k-1}^{(i)}, \quad i = 1, ..., N, \quad (15)$$

where $v_{k-1}^{(i)}$ is a draw from the distribution with PDF $p(v_{k-1})$. The measurement $y_k$ is then used to update the weights by

$$w_k^{(i)} = w_{k-1}^{(i)}p(y_k|x_k^{(i)}), \quad i = 1, ..., N. \quad (16)$$

This yields the particle set $S_k = \{x_k^{(i)}, w_k^{(i)}\}_{i=1}^N$ at time step $k$. By iterating (15) and (16), a sample from $p(x_k|y_{1:k})$ is thus recursively obtained. The recursion is initialized by making $N$ draws from an a initial distribution with PDF $p(x_0)$. It can be shown that, as formulated for now, the variance of the particle set can only increase over time, with the consequence that the weights of all particles except one will approach zero as $k \rightarrow \infty$ [7]. When this happens, the filtering has broken down to a pure simulation of the system. To remedy this problem and concentrate the particles to the domain of interest, i.e. where the density of $p(x_k|y_{1:k})$ is high, resampling can be performed. In the resampling step, a new set of particles $S'_k = \{x_k^{(i)}, w_k^{(i)}\}_{i=1}^N$ is created and replaces the old particle set $S_k$. Bootstrapping is a common approach where a new set of particles $S'_k$ is created by making $N$ draws with replacement from the old particle set such that $Pr(x_k^{(i)} = x_k^{(j)}) = w_k^{(i)}$ and setting $w_k^{(i)} = 1/N$. An illustration of how the particle set evolves during the prediction, update and resampling step, is given in Fig. 3.

5.3 Grid-based methods

Grid-based methods, see e.g. [4], also apply to nonlinear systems with general noise distributions. The involved PDFs are approximated by a set of point masses over a discrete set of grid points $\{x_k^{(i)}\}_{i=1}^N$, $x_k^{(i)} \in \mathbb{R}^n$, with associated weights $\{w_{k|i}^{(i)}\}_{i=1}^N$. The PDF $p(x_{k-1}|y_{1:k-1})$ is then approximated as

$$p(x_{k-1}|y_{1:k-1}) \approx \sum_{i=1}^N w_{k-1|i-1}^{(i)} \delta(x_{k-1} - x_{k-1}^{(i)}), \quad (17)$$
Figure 3: The evolution of a set of particles. The left part of the figure shows the particles as dots and their weights are represented by the dot sizes. The right part of the figure shows the discrete weighted estimate $\hat{p}(x_n|y_{1:n})$ of $p(x_n|y_{1:n})$ given by the particles. Step (1), (2), (3) and (4) shows the initial set $S_{k-1|k-1}$, the propagated set $S_{k|k-1}$, the updated set $S_{k|k}$ and the resampled set $S'_{k|k}$ respectively.

where the approximation sign here should be interpreted as that the weighted set of point masses carries approximately the same statistical information about the state as the true PDF, such as e.g. the mean and variance. The $i$-th weight is propagated via the prediction and update equations as

$$w^{(i)}_{k|k-1} = \sum_{j=1}^{N} w^{(j)}_{k-1|k-1} p(x^{(i)}_k|x^{(j)}_{k-1}),$$  

$$w^{(i)}_{k|k} = w^{(i)}_{k|k-1} p(y_k|x^{(i)}_{k}).$$  

and the predicted PDF $p(x_k|y_{1:k-1})$ and updated PDF $p(x_k|y_{1:k})$ are approximated by

$$p(x_k|y_{1:k-1}) \approx \sum_{i=1}^{N} w^{(i)}_{k|k-1} \delta(x_{k-1} - x^{(i)}_{k-1}),$$

$$p(x_k|y_{1:k}) \approx \sum_{i=1}^{N} w^{(i)}_{k|k} \delta(x_{k} - x^{(i)}_{k}).$$

A problem with grid-based methods is a large computational burden associated with it. To achieve satisfactory accuracy, a large number of grid points must be used. As the number of grid points grows exponentially with the dimension of the problem, its usability is confined to low-dimensional problems.
5.4 Computational complexity

The computational complexity of the optimal solution to the recursive Bayesian estimation problem is a major obstacle in real-time applications as well as in high-dimensional problems. In the linear Gaussian case (KF), the computational complexity grows as $O(n^3)$. However, for the grid-based methods, the complexity grows exponentially with the dimension. If one dimension requires $N$ grid points, then a two-dimensional approximation requires $N^2$ grid points, and a $n$-dimensional problem requires $N^n$ grid points. This is known as the curse of dimensionality and poses a severe problem, limiting the applicability of the method to low-dimensional cases. Monte-Carlo methods have the appealing property that the convergence rate of the estimate is independent of the dimension of the problem [5]. However, the number of particles required for a satisfactory estimation accuracy is typically fairly large that makes the method computationally expensive.
6 Multicore architecture

6.1 Evolution of the multicore processor

A multicore processor is a single computing component with two or more independent actual processors, called cores or central processing units (CPUs), which are the units that read and execute program instructions. The material in this section is based on the references [25], [17], [10], [26], [16]. For decades, it was possible to improve the performance of the single core CPU by shrinking the area of the integrated circuit and increasing the clock rate at which it operated. In the early 2000’s, the rate of increase of the computational power for the single core processor began to stall, mainly due to three major bottlenecks:

- The memory wall; Over decades, processor speeds have increased at far faster rates than the memory speeds. As the memory system cannot deliver data fast enough to keep the processor busy, the memory has become a bottleneck to performance improvement of the sequential processor.

- The power wall; The power consumption of a processor increases exponentially with each factorial increase of operating frequency. Hence, it is not possible, due to both power and heat dissipation concerns, to improve the performance of a single core processor by increasing the operating frequency.

- The ILP wall; An increasing difficulty of finding enough instruction level parallelism in a single instructions stream to keep a high-performance single-core processor busy.

In the pursue of improving the computational capacity of a system, more focus was put on parallel architectures that have started to evolve at an increasing rate and become a standard piece of hardware. Any PC, and many mobile phones bought today, have two or more processors on an integrated circuit. The purpose of fitting several cores on the same chip is mainly to improve the computational capacity of the system. Another aspect, important to low power consumption systems such as e.g. battery driven mobile devices and low power communication systems, is the lower power-per-FLOP ratio provided by parallel hardware. Several cores on the same chip generally consume less power than the same amount of cores located on different chips.

6.2 Parallel architectures

There is a plethora of parallel processing architectures with examples as shared memory multicores (SMCs), graphical processing units (GPUs), and computer clusters. Roughly speaking, all parallel architectures can be modeled as $M$ processing units with some kind of interconnection, each having a private memory.
Figure 4: A simplified picture of a shared memory multicore architecture.

and possibly connected to a shared memory. What differs between the architectures are the sizes of the shared and private memory, the interconnection topology and the bandwidth of the interconnection. How well a parallelization executes depends on very much how well it maps to the particular architecture used.

In a computer cluster, processors located at different chips are interconnected over a communication network. The bandwidth of the network is typically relatively low, there is no shared memory but the private memory is relatively large. A GPU is, as its name suggests, mainly constructed to process computer graphics. It is suitable when the same operation is to be performed on several independent data streams. A GPU architecture has a large memory bandwidth, no private memory, and a medium-sized shared memory. A GPU can have hundreds of processors, but is only suitable for a narrow class of problems that can provide the amount of fine-grained parallelism required for efficient operation.

This thesis is mainly concerned with the shared memory multicore architecture (SMC) that is a flexible architecture suited for embedded real-time applications. The term multicore refers to a processor where several CPUs are manufactured on the same integrated circuit die. Fig. 4 shows a simplified picture of a SMC. The CPUs are connected to a shared memory (the RAM) via a shared bus. In addition to the shared memory, each processor has a private memory, the cache, to which only that particular CPU has access. In general, most SMCs have several layers of cache where some levels of the cache are shared among two or more cores. However, to understand the concept and reasoning of a multicore implementation, it many times suffices to envisage the simplified description with a single cache per processor. The CPUs can operate independently of each other, and the interprocessor communication is accomplished through reads and writes to the shared memory.
7 Parallel implementation

7.1 Parallel implementation

The performance improvement gained by the use of a multicore processor depends very much on the software algorithms used and their implementation. Ideally, an implementation may realize speedup factors near the number of cores used. Most applications, however, are not accelerated so much unless programmers invest an amount of effort in re-factoring the whole problem. To design well-performing software that scales well and executes fast, it is important to understand the basics of the architecture on which the software is intended to execute.

Roughly speaking, when designing a parallel implementation, it is sought to determine a number of tasks that can execute as large portions of work as possible with a minimal amount of interaction. It is though important to remember that parallelization is not a goal by itself, but merely a way of improving the execution time. Constructing an implementation that runs perfectly in parallel, but slower than the sequential version of the implementation presents nothing of interest.

Parallelization can provide many benefits for an implementation. The programming and debugging of a parallel program can though be much more challenging than for a sequential program. Constructing and debugging parallel code to ensure its correctness can be a difficult task. Some examples of problems that a parallel programmer must deal with that are not present in sequential programming are:

- **Parallel overhead** The amount of time required to coordinate parallel threads, as opposed to doing useful work. Parallel overhead can include factors such as thread start-up time, synchronization, software overhead imposed by parallel compilers, libraries, tools, operating system, thread termination time, etc.

- **Load balancing** For an implementation to execute efficiently, the workload of the processing units should be as balanced as possible, i.e. each processor should have an equal amount of computations to perform. With an unbalanced workload, one or more processors will be idle waiting for the more loaded processors to finish, and thereby wasting computational capacity of the system.

- **Cache coherency** On a multicore computer, several processors can have the same piece of data in their private caches. If one processor modifies that data, the other processors must be notified about this to get a consistent view of the memory. How this scheme is adopted is architecture-dependent.

- **Synchronization** At some points in the execution, two or more of the processing units must be synchronized, i.e. they must wait at some point to
make sure that the other processors have reached a certain point in the execution stream.

**Communication** In order to complete a task, processors must communicate with each other. How and when to communicate must be specified by the programmer.

**Race conditions** If two or more processors are accessing the same piece of data, the outcome of the program can be inconsistent, depending on in which order the processors happened to read and modify the shared data. To prevent this, mutual exclusion mechanisms must be used to ensure correct results.

For a more extensive exposition of parallel programming issues see e.g. [25], [26], [16].

When designing a parallel program, the procedure can be partitioned into three stages, see e.g. [18]:

- **Partitioning**: Opportunities for parallel execution are exposed. A fine-grained decomposition of the problem is created, where a large number of tasks that can be executed concurrently are identified.

- **Communication**: The communication required among the fine-grained tasks identified in the partitioning stage is explored.

- **Agglomeration**: It is determined how to agglomerate tasks identified by the partitioning phase, so as to provide a smaller number of tasks which can execute concurrently with a small amount of interaction. It is also determined if data and/or computation should be replicated in order to minimize interaction between tasks.

Automatic parallelization has been a research topic for several decades. Yet fully automatic parallelization of sequential programs by compilers still remains a challenge due to its need for complex program analysis and dependance on unknown factors, such as input data range, during compilation. In most cases, to parallelize other than embarrassingly parallel algorithms, insight and understanding to the theory of the underlying algorithm is required.

### 7.2 Software

There are many different languages available for multicore programming, with such examples as Cilk++, OpenMP, OpenHMPP, FastFlow, Skandium, and MPI. OpenMP ¹ has been the choice for all developed algorithms in this thesis because of the algorithms’ suitable mapping to the fork join model adopted by OpenMP. OpenMP is a collection of directives, library routines and environment variables that may be used to parallelize Fortran, C and C++ programs.

¹http://openmp.org
Figure 5: Fork join model. A program containing three sequential sections of work, $S_1$, $S_2$, and $S_3$, and two parallel sections with the tasks $A_1$, $B_1$, $C_1$ and $D_1$ in the first section and $A_2$ and $B_2$ in the second section. The master thread is marked with gray.

for execution on shared memory platforms. A master thread running sequentially forks a specified number of slave threads with tasks divided among them. The slave threads then run in parallel and the runtime environment allocates threads to different processing units. After the execution of the parallelized code, the threads join back into the master thread that continues onward to the end of the program. See Fig. 5 for an illustration of the flow of execution using OpenMP. Both task parallelism and data parallelism can be achieved using OpenMP in this way.

7.3 Performance measures

The execution time of a parallel implementation is often the most important performance measure, as in many cases the parallelization is performed in order to shorten it. How the execution time scales with the number of cores is also of importance as it specifies how much the execution time can be improved by employing more processing units. This is characterized by the speedup $s(M)$ defined as

$$s(M) = \frac{t(1)}{t(M)},$$

where $t(1)$ is the execution time of the fastest possible (known) sequential implementation and $t(M)$ is the execution time of the program using $M$ cores. The efficiency, specifying how well the computational resources are utilized, is calculated as

$$e(M) = \frac{s(M)}{M}.$$

An ideal speedup curve is linear in the number of processors used, i.e. $s(M) = M$, and has the efficiency $e(M) = 1$. It is actually possible to achieve a speedup slightly above linear, known as superlinear speedup. This phenomenon can occur because when more processors are employed, the size of the private
cache increases allowing a better cache performance that can result in an efficiency over 1.

A simplified formula for the speedup can be obtained as follows. Assume that \( p_1 \) and \( p_{||} \) are the portions of the program that are executed sequentially and in parallel, respectively. The parallel overhead is denoted with \( c(M) \). The execution time on \( M \) processors is given by

\[
t(M) = p_1 + \frac{p_{||}}{M} + c(M).
\]

Noting that \( C(1) = 0 \), the speedup is obtained from

\[
s(M) = \frac{t(1)}{t(M)} = \frac{p_1 + p_{||}}{p_1 + \frac{p_{||}}{M} + c(M)} = \frac{1}{p_1 + \frac{p_{||}}{M} + c(M)}.
\]

Ignoring the overhead term, this formula is known as Amdahl’s law [1]. A consequence of it is that the highest achievable speedup, assuming no parallel overhead, is given by

\[
s(\infty) = \frac{1}{p_1}.
\]

Hence, a program having e.g. \( p_1 = 0.1 \) can never reach a greater speedup than \( 1/0.1 = 10 \) times, no matter how many processors used. It is therefore of utmost importance to keep the sequentially executed part of an implementation as small as possible. Fig. 6 shows speedup curves for different values of \( p_1 \), where the parallel overhead is given by \( c(M) = 0.01 + 0.01 \cdot M^2 \). As \( c(M) \) increases with an increasing number of processors, the speedup curve has a maximum. Obviously, it is not beneficial to increase the number of processors beyond this maximum since the overhead becomes too large and increases the execution time. The figure also shows the characteristics of a speedup curve when a bottleneck, such as memory bandwidth, has been hit.

### 7.4 Data partitioning

Another important issue in parallel programming is how to partition the data among the cores. Ideally, each core should touch as small amount of data as possible, in order to not saturate the memory bus, and also the data it touches should preferably be local to that core, to minimize the intercore communication. Some of the concepts are exemplified by matrix operations below.

Let \( A, B, \) and \( C \) denote \( n \times n \) matrices, and \( x, y, z \) denote column vectors of length \( n \). Assume a matrix matrix multiplication \( C = AB \) is to be parallelized.
Figure 6: Speedup curves for a program with different portions of sequentially executed code, \( p_i \), and speedup curve for an implementation that has hit a bottleneck such as saturation of the memory bus. For reference linear speed up is marked by the dashed line.

Let \( \sqrt{M} \) be integer and consider a partitioning

\[
AB = \begin{bmatrix}
A_1 & A_2 & \cdots & A_{\sqrt{M}} \\
B_1 & B_2 & \cdots & B_{\sqrt{M}} \\
A_1B_1 & A_1B_2 & \cdots & A_1B_{\sqrt{M}} \\
A_2B_1 & A_2B_2 & \cdots & A_2B_{\sqrt{M}} \\
& & \ddots & \ddots \\
A_{\sqrt{M}}B_1 & \cdots & \cdots & A_{\sqrt{M}}B_{\sqrt{M}}
\end{bmatrix}
\]

where one processor computes one of the \( M \) blocks \( A_iB_j \), \( 1 \leq i,j \leq \sqrt{M} \).

Compare this to the partitioning

\[
AB = \begin{bmatrix}
A_1 & A_2 & \cdots & A_M \\
B & B & \cdots & B \\
A_1B & A_2B & \cdots & A_MB
\end{bmatrix}
\]

where one processor computes one of the blocks \( A_iB \), \( 1 \leq i \leq M \). For both partitionings the workload is perfectly distributed among the processors. All processors will perform an equal amount of computations and no computations are
duplicated. However, for partitioning (20), each processor must touch $\frac{M+1}{M}n^2$ data elements while for partitioning (21) each processor must only touch $\frac{2}{M}n^2$ elements. It is clearly beneficial to use partitioning (21), for $M > 2$, in order to touch as small amount of data as possible. Specifically, it is seen that the total amount of the data touched is given by $(M + 1)n^2$ and $2n^2$ elements by partitioning as in (20) and in (21), respectively. The amount of data touched thus increases linearly with the number of processors, $M$ when using partitioning in (20), while being constant for partitioning in (21). Thus, using partitioning (20), one could not expect an implementation to scale well for a large number of processors since the memory bus will eventually be strained and limit the speedup.

As another example, consider a sequence of matrix computations

$$y = Ax, \quad C = yz^T,$$

that are to be parallelized. Compare a partitioning

$$\begin{bmatrix} y_1 & y_2 & \cdots & y_M \end{bmatrix}^T = \begin{bmatrix} A_1 & A_2 & \cdots & A_M \end{bmatrix}^T x,$$

$$\begin{bmatrix} C_1 & C_2 & \cdots & C_M \end{bmatrix}^T = \begin{bmatrix} y_1 & y_2 & \cdots & y_M \end{bmatrix}^T z^T,$$  \hspace{1cm} (22)

where processor $i$ computes $y_i = A_i^T x$, $C_i = y_i z^T$ with a partitioning

$$y_1 + y_2 + \ldots + y_M = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_M \end{bmatrix},$$

$$\begin{bmatrix} C_1 & C_2 & \cdots & C_M \end{bmatrix}^T = \begin{bmatrix} y_1 & y_2 & \cdots & y_M \end{bmatrix}^T z^T,$$  \hspace{1cm} (23)

where processor $i$ computes $y_i = A_i x_i$, $C_i = y_i z^T$. For both partitionings, each processor will perform the same number of computations and touch the same amount of data. However, the partitioning in (23) requires a synchronization point in between the two lines, since the partial sums must be added to form $y$, i.e. $y = \sum_{m=1}^{M} y_m$. Partitioning (22) thus provides better potential for an efficient implementation since the processors can perform larger amount of work independently of each other without synchronization and communication.

### 7.5 High-performance computing

Since one of the main points in using parallel hardware is to achieve faster execution times, it is not only important that the computations are made in parallel, but also that the program is optimized w.r.t. the execution time.
Today’s compilers can automatically do significant optimizations to the code. However, to achieve high performance, the programmer must still invest an effort in optimization of the code, giving the compiler a good ground to work on. Some of the optimizations are discussed here and can be found in [15], [9]. One of the most important aspects in achieving fast execution is to handle the memory accesses efficiently. A program, where memory access has been optimized, can potentially execute several orders of magnitude faster than an unoptimized program. Two important data access properties exhibited by many programs are spatial and temporal locality. This is something that the cache memory utilizes, at a hardware level, in the following manner.

- **Spatial locality**: If an element at a specific memory address is accessed, it is likely that data elements at nearby addresses will be accessed soon. Therefore, neighboring data elements to the one that are being accessed now will also be brought to the cache.

- **Temporal locality**: If an element is being accessed now, it is likely that it will be used soon again. Therefore, the cache will keep the most recently used data.

Thus, when a data element is brought to the cache, not only that particular element, but also the neighboring data elements will be brought to the cache. How many elements are brought in depends on the cache line size. As it is time consuming to move data in the memory, it will benefit performance greatly if the code were written in a such manner that the data movement from the main memory to the CPUs were minimized. When a data element is brought to the cache, it is hence desirable to use the element to accomplish as many calculations as possible that the element is involved in, before it is thrown out of the cache. Cache re-use can be an even more critical issue on multicore processors than on single core processors due to their larger computational power and more complex memory hierarchies.

Further considerations when designing high performance software are the hardware-specific optimizations. As an example, a pre-fetcher is a hardware mechanism that tries to predict which data will be used in a near future and fetch this piece of data into the cache as to be readily available when requested from the processor. A programmer aware of such mechanisms can utilize them to improve the performance substantially.

Loop unrolling, loop fusion, hoisting and avoiding branching are other techniques to improve the execution time, to mention some.

As linear algebra operations are commonly encountered in high-performance computing, optimized libraries such as BLAS (Basic Linear Algebra Subprograms) \(^2\), have been developed for these type of operations. These libraries are extremely efficiently implemented, utilizing hardware-specific optimizations for particular architectures. The set of routines is though limited and covers only the more basic operations.

\(^2\)www.netlib.org/blas/
8 Parallel implementation of filtering methods

In this section, it is discussed how the estimation methods described in Sec. 5 can be implemented in parallel, which is the main subject of this thesis. Parallelizations of the Kalman filter, particle filter, and of a novel approach based on orthogonal series expansions have been developed and presented in Papers I-IV. Parallelizations of the static KF, the UKF and grid-based methods have also been performed but are not presented in any paper. An exposition of the parallelization approaches and results is given below. In general, it has been found that the solution methods to the recursive Bayesian estimation problem are amenable to parallel implementation, as substantial speedup has been achieved for implementation of all methods.

In the sequel, \( M \) denotes the number of processors employed for the execution. As before \( n \) and \( p \) denote the dimension of the state space and the dimension of the output, respectively. For the grid-based methods, \( N \) denotes the number of grid points and for the Monte Carlo-based methods, it means the number of particles used.

8.1 Kalman filter

8.1.1 General Kalman filter

A parallelization of the time-varying Kalman filter is given in Paper II. The parallelization utilizes a non-restrictive assumption that the system matrix of the plant has a banded structure. The parallelization method is especially efficient in the parameter estimation case that is studied in detail in Paper I.

8.1.2 Static Kalman filter

As the static Kalman filter given by (12) simply consists of a single line made up of matrix and vector multiplications, the most efficient way of implementing this is by using a multi-threaded optimized linear algebra library such as e.g. BLAS. A BLAS-based implementation has been performed and the speedup curves from the execution are presented in Fig. 8. The gained speedup varies significantly depending on the problem size. A discussion regarding this is given below.

As the work-overhead ratio increases with an increasing problem size, a better scalability is obtained for the values of \( n \) up to \( n = 1000 \). As a consequence of memory bandwidth saturation, the speedup drops drastically for larger \( n \).

The hardware used provides 16MB of cache. A double precision \( n \times n \) matrix requires \( 8n^2 \) bytes of memory. Solving the equation \( 16MB = 8n^2B \) yields \( n \approx 1400 \). Hence, for \( n > 1400 \), the matrix will not fit into the cache and has to be brought from the main memory on every iteration, in which case the memory bandwidth becomes a bottleneck.

To understand the dip in the speedup curve for \( n = 1000 \) in Fig. 7, a more detailed model of the architecture has to be employed. The machine on which
Figure 7: Memory connections for octa-core, consisting of two quad-core CPUs. Memory is given by the gray blocks, the CPUs are marked by black dots.

Figure 8: Speedup curves for parallel execution of static Kalman filter for different problem sizes $n$. For reference linear speedup is marked by the dashed line.

the code is executed provides 8 cores, but consists of two Nahalem Quad Core, each having 8 MB cache, as shown in Fig. 7. The first 4 threads have been scheduled to run on the one quad core, and threads 5 to 8 have been scheduled to run on the other quad core. When 4 cores are used, there will be only 8MB cache available, in which case the data of $1000^2 \cdot 8 = 8MB$ indeed will not fit as some space is occupied by other data. However, using more than 4 cores, 16 MB of data will be available and the data will fit into the cache and hence the program will not be limited by the memory bandwidth. The situation can of course be resolved by scheduling threads 1 and 2 to run on one of the quad cores, and thread 3 and 4 on the other, in which case the dip would disappear. However, the scheduling has been kept as it is to demonstrate a phenomenon that gives insight into the problems that can occur in a parallel implementation. Note though that the execution times are very low.
Table 1: Single core execution time, $T$ for different problem sizes $n$, for execution of static Kalman filter.

<table>
<thead>
<tr>
<th>$n$</th>
<th>100</th>
<th>200</th>
<th>500</th>
<th>1000</th>
<th>1500</th>
<th>2000</th>
<th>5000</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T$ [ms]</td>
<td>0.0038</td>
<td>0.0169</td>
<td>0.1068</td>
<td>0.5250</td>
<td>1.6830</td>
<td>2.9590</td>
<td>18.1081</td>
</tr>
</tbody>
</table>

### 8.1.3 Extended Kalman filter

The extended Kalman filter (EKF) is based on the same computations as the Kalman filter, with an additional linearization step (13), (14). The linearization step can be completely parallelized. Assume that $n/M$ and $p/M$ are integer and let

$$
\begin{align*}
\mathbf{f}_{k,i} &= \left[ f_{\frac{n}{M}(i-1)+1}(\mathbf{x}) \ f_{\frac{n}{M}(i-1)+2}(\mathbf{x}) \ \cdots \ f_{\frac{n}{M}i}(\mathbf{x}) \right]^T, \\
\mathbf{h}_{k,i} &= \left[ h_{\frac{p}{M}(i-1)+1}(\mathbf{x}) \ h_{\frac{p}{M}(i-1)+2}(\mathbf{x}) \ \cdots \ h_{\frac{p}{M}i}(\mathbf{x}) \right]^T.
\end{align*}
$$

Processor $i$ will then compute

$$
\begin{align*}
\mathbf{F}_{k,i} &= \left. \frac{\partial \mathbf{f}_i(\mathbf{x})}{\partial \mathbf{x}} \right|_{\mathbf{x}=\mathbf{x}_{k-1|k-1}}, \\
\mathbf{H}_{k,i} &= \left. \frac{\partial \mathbf{h}_i(\mathbf{x})}{\partial \mathbf{x}} \right|_{\mathbf{x}=\mathbf{x}_{k|k-1}},
\end{align*}
$$

and the complete Jacobians are given by

$$
\mathbf{F}_k = \begin{bmatrix} \mathbf{F}_{k,1} \\ \mathbf{F}_{k,2} \\ \vdots \\ \mathbf{F}_{k,M} \end{bmatrix}, \quad \mathbf{H}_k = \begin{bmatrix} \mathbf{H}_{k,1} \\ \mathbf{H}_{k,2} \\ \vdots \\ \mathbf{H}_{k,M} \end{bmatrix}.
$$

Provided that the matrix $\mathbf{F}_k$ possesses a banded structure, the same method as for parallelization of the original Kalman filter can then be applied to the linearized system. This might be a restrictive assumption while an important special case occurs when the Jacobians are sparse matrices with zeros located at the same positions at each time step $k$. For this case, a transformation that optimizes the band structure of the matrices can be applied as discussed in Paper II.

### 8.1.4 UKF

Parallelizing the UKF is a matter of parallelizing the Cholesky factorization. An implementation of the UKF, based on the BLAS routine for Cholesky factorization, has been performed. The execution time and speedup results are summarized in Tab. 2 and Fig. 9, respectively. As can be seen, the scalability of the parallel UKF is good when the problem size is large ($n \gtrsim 1000$).
Figure 9: Speedup curves for parallel implementation of UKF for different problem sizes $n$. For reference linear speedup is marked by the dashed line.

Table 2: Single core execution time, $T$ for different problem sizes $n$, for execution of UKF.

<table>
<thead>
<tr>
<th>$n$</th>
<th>100</th>
<th>200</th>
<th>500</th>
<th>1000</th>
<th>2000</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T$ [ms]</td>
<td>0.0958</td>
<td>0.4911</td>
<td>6.2881</td>
<td>45.1539</td>
<td>336.1521</td>
</tr>
</tbody>
</table>

For smaller problem sizes, the parallel overhead constitutes a disproportionally large part of the execution time, which property results in a poor scalability.

### 8.2 Grid-based methods

The fact that the algorithm for grid-based methods consists of summation of independent elements make them especially suitable for parallel implementation. One iteration of a grid-based method consist of computing (18) and (19), i.e.

$$w_{k|k-1}^{(i)} = \sum_{j=1}^{N} w_{k-1|k-1}^{(j)} p(x_{k}^{(i)}|x_{k-1}^{(j)}),$$

$$w_{k|k}^{(i)} = w_{k|k-1}^{(i)} p(y_{k}|x_{k}^{(i)}).$$

$i = 1, 2, ..., N$. Defining
Algorithm 1 Pseudo code for one iteration of a grid based method.

- for $i=1:N$
  - $w_{k|k-1}^{(i)} = 0$
  - for $j=1:N$
    * $w_{k|k-1}^{(i)} = w_{k|k-1}^{(i)} + w_{k-1|k-1}^{(j)} p(x_k^{(i)} | x_k^{(j)})$
    - end
  - $w_{k|k}^{(i)} = p(y_k | x_k^{(i)}) w_{k|k-1}^{(i)}$
- end

\[
\begin{align*}
\mathbf{w}_{k|k-1} &= \begin{bmatrix}
  w_{k|k-1}^{(1)} & w_{k|k-1}^{(2)} & \cdots & w_{k|k-1}^{(N)} \\
p(x_k^{(1)}, x_k^{(1)}) & p(x_k^{(1)}, x_k^{(2)}) & \cdots & p(x_k^{(1)}, x_k^{(N)}) \\
p(x_k^{(2)}, x_k^{(1)}) & p(x_k^{(2)}, x_k^{(2)}) & \cdots & p(x_k^{(2)}, x_k^{(N)}) \\
\vdots & \vdots & \ddots & \vdots \\
p(x_k^{(N)}, x_k^{(1)}) & \cdots & \cdots & p(x_k^{(N)}, x_k^{(N)}) \\
p(y_k, x_k^{(1)}) & 0 & \cdots & 0 \\
0 & p(y_k, x_k^{(2)}) & \ddots & \vdots \\
\vdots & \vdots & \ddots & 0 \\
0 & \cdots & 0 & p(y_k, x_k^{(N)})
\end{bmatrix}^T,
\end{align*}
\]

\[
\begin{align*}
\mathbf{P}_k &= \begin{bmatrix}
p(x_k^{(1)}, x_k^{(1)}) & p(x_k^{(1)}, x_k^{(2)}) & \cdots & p(x_k^{(1)}, x_k^{(N)}) \\
p(x_k^{(2)}, x_k^{(1)}) & p(x_k^{(2)}, x_k^{(2)}) & \cdots & p(x_k^{(2)}, x_k^{(N)}) \\
\vdots & \vdots & \ddots & \vdots \\
p(x_k^{(N)}, x_k^{(1)}) & \cdots & \cdots & p(x_k^{(N)}, x_k^{(N)}) \\
p(y_k, x_k^{(1)}) & 0 & \cdots & 0 \\
0 & p(y_k, x_k^{(2)}) & \ddots & \vdots \\
\vdots & \vdots & \ddots & 0 \\
0 & \cdots & 0 & p(y_k, x_k^{(N)})
\end{bmatrix},
\end{align*}
\]

\[
\begin{align*}
\mathbf{Q}_k &= \begin{bmatrix}
p(y_k, x_k^{(1)}) & 0 & \cdots & 0 \\
0 & p(y_k, x_k^{(2)}) & \ddots & \vdots \\
\vdots & \vdots & \ddots & 0 \\
0 & \cdots & 0 & p(y_k, x_k^{(N)})
\end{bmatrix},
\end{align*}
\]

the prediction and update equations (18) and (19) can be expressed in matrix form as

\[
\begin{align*}
\mathbf{w}_{k|k-1} &= \mathbf{P}_k \mathbf{w}_{k-1|k-1}, \\
\mathbf{w}_{k|k} &= \mathbf{Q}_k \mathbf{w}_{k|k-1}.
\end{align*}
\]

These matrix equations could then be implemented using, e.g. BLAS or by the pseudo code in Alg. 1, where the parallelization is performed over the $i$ loop iterations.

Tab. 3 and Fig. 10 show the execution time and, respectively, scalability, of an implementation of Alg. 1, for different problem sizes $N$. Note that the problem size is given by the number of grid points $N$ and not by the number of states $n$ as for the Kalman filter based methods.

30
Figure 10: Speedup curves for execution of Alg. 1 for different problem sizes N. For reference linear speedup is marked by the dashed line.

Table 3: Execution time, $T$ for sequential execution of grid based estimator for different number of grid points $N$.

<table>
<thead>
<tr>
<th>$N$</th>
<th>100</th>
<th>200</th>
<th>500</th>
<th>5000</th>
<th>20000</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T$ [ms]</td>
<td>0.1109</td>
<td>0.4389</td>
<td>2.7909</td>
<td>270.7901</td>
<td>4334.2</td>
</tr>
</tbody>
</table>

8.3 Particle filter

In the particle filter, the particles are mutually independent in the prediction and update steps (15) and (16) that are hence relatively simple to parallelize. However, in the resampling step, the resampled set depends on the complete particle set, which fact makes it less amenable to parallelization. An investigation of parallelization methods is given in Paper I.

8.4 Series expansion

A novel solution method to the recursive Bayesian estimation problem has been developed, with parallelization in mind. The method uses orthogonal series expansions of the PDFs involved in the recursive estimation problem. Via the prediction and update steps (3), (4), the coefficients of the expansions are propagated. As it is possible to approximate a PDF with far fewer coefficients using series expansions, than e.g. approximating it over a grid or by a random sample, the computational burden of the developed method is substantially lower than that for grid-based and Monte-Carlo-based methods. The orthogonality properties of the basis functions tend to separate the workload into independent segments and the resulting algorithm possesses thereby very favorable parallelization properties. The method is given in Paper III.
9 Concluding remarks

The main topic of this licentiate thesis is the parallelization of filtering methods. It has been found that several of the optimal filtering methods are suitable for parallel implementation, in certain ranges of problem sizes. It is concluded from the experiments that substantial improvements can be achieved by making "tailor"-made parallelization, compared to straightforward implementations based on multi-threaded libraries. For many of the suggested parallelizations, a linear speedup in the number of cores has been achieved that have provided up to 8 times speedup on a double quad-core computer. As the evolution of the parallel computer architectures is unfolding rapidly, many more processors on the same chip will become available. The developed methods do not, of course, scale infinitely, but definitely can exploit and harness some of the computational power of the next generation of parallel platforms, allowing for optimal state estimation in real-time applications.
Efficient Parallel Implementation of State Estimation Algorithms on Multicore Platforms

Olov Rosén, Alexander Medvedev*†

Abstract

For many applications in signal processing and control it is crucial that estimates of the state vector in a dynamic system can be obtained in real time. This poses the problem of producing algorithms that are fast enough to enable on-line execution. In this article, it is investigated how two of the most popular and powerful state estimation algorithms, the Kalman filter and the particle filter, can be efficiently implemented in parallel on a multicore architecture. The proposed parallel implementations are analyzed in terms of hardware requirements, such as memory bandwidth and available cache memory, to provide the desired speedup. The algorithms are exemplified by and evaluated in an adaptive filtering and a bearings-only tracking application. In the cases when original algorithms have been modified for parallelization, the accuracy of the estimates obtained is evaluated in comparison with that of the sequential algorithm. It is found that linear speedup, in the number of cores used, can indeed be achieved without loss of accuracy, for both state estimation algorithms.

1 Introduction

State estimation in dynamic systems from uncertain measurements is a fundamental task in control and signal processing. Given a mathematical model of the system in a stochastic framework, the problem can be formulated as to find a maximum likelihood estimate of the state vector and subsequently solved by recursive Bayesian estimation.

If the underlying system is linear with white process and measurement noise, an analytical solution to the estimation problem can be found in the form of the

---

*The authors are partially supported by the project "Computationally Demanding Real-Time Applications on Multicore Platforms" funded by Swedish Foundation for Strategic Research.

†O. Rosén and A. Medvedev is with the Institution of Information Technology, Department of System and Control, Uppsala University, Uppsala SE-751 05, Sweden {olov.rosen, alexander.medvedev}@it.uu.se
Kalman filter (KF). However, if the system is nonlinear or the disturbances do not fulfill the (filtered) white noise assumption, the problem in general becomes analytically intractable and calls for approximative approaches. Examples of these are point mass filters, numerical integration methods, Monte Carlo methods, to name a few, see e.g. [1] for an overview of the area.

Monte Carlo methods such as the particle filter (PF) (also known as the sequential Monte Carlo algorithm) [2, 3, 4] present an attractive approximative solution to the problem of state estimation due to their ability to handle nonlinear systems with arbitrary noise distributions. Furthermore, PF does not suffer from that the computational burden grows exponentially with the dimension of the problem (curse of dimensionality), as point mass filters and numerical integration methods do.

For linear systems of moderate dimension, solving the recursive Bayesian estimation problem does not require a large amount of computations. However, for nonlinear systems or linear systems of large dimension, it can become very computationally expensive. As many applications require on-line state estimation in hard real time, there is impetus to shorten the necessary computational time of the popular recursive estimation algorithms.

The computer industry has now entered the multicore era with hardware computational capacity increased by adding more processors (cores) on one chip. All major processor chips manufactures have moved to a multicore (many core) design and sequential processors will not be available already in near future [5]. Almost any personal computer and some cellular phones bought today contain two or more processors, and the number of processors is predicted to drastically increase in the near future. The state-of-the-art hexa-core processors contain six cores (e.g. AMD Phenom II X6, Intel Core i7 Extreme Edition 980X) [6, 7]. Naturally, due to huge performance, lower energy consumption and heat dissipation, multicore architectures are highly appealing for computationally demanding embedded control and signal processing applications. Alas, in most cases, streamlined real-time software runs slower on a multicore computer than on a single core one with the same clock frequency. This poses a major problem to companies dependent on hard real time systems: they actually face a slowdown of their software due to the oncoming departure of sequential processors.

Parallelization of algorithms per se does not automatically mean that they can be run efficiently on multicore platforms. There are plenty of examples where "embarrassingly parallel" algorithms have resulted in applications running even slower in parallel than sequentially. A real challenge to multicore software is presented by so-called "memory wall" that is a disparity between how fast a CPU can operate on data and how fast it can get data. In order to achieve speedup on multicore computers, parallel algorithms must be optimized with respect to the cache memory use so that they do not overcome the available on the present architecture memory bandwidth.

In this article parallel implementation of the KF and the PF is studied. The speedup gained from an implementation on a multicore architecture using up
to eight cores is evaluated, as well as the influence of parallelization on the estimation accuracy. For KF, a typical to adaptive filtering single-input single-output case is studied, where a reordering of the equations allows for a parallel implementation that is capable of achieving linear speedup in the number of cores used. For PF, four existing parallel particle filters, Globally Distributed Particle Filter (GDPF) [8], Resampling with Proportional Allocation filter (RPA) [9], Resampling with Non Proportional Allocation filter (RNA) [9] and the Gaussian Particle Filter (GPF) [10] are implemented and evaluated for a test bed solving a bearings-only tracking problem.

After introducing necessary nomenclature in Section 2, Recursive Bayesian estimation is briefly reviewed in Section 3. The article aims at bringing computer architecture insights into the design of signal processing algorithms and, therefore, a summary of the memory architecture for a multicore is provided in Section 4, to facilitate the understanding of some points in the implementation. For a more extensive exposition of these concepts see e.g. [11]. In Section 5, theoretically expected speedup from a parallel implementation is calculated. In Section 7 and Section 12, the parallel implementation and performance evaluation, together with a relevant discussion, are given for KF and PF, respectively. Finally the conclusions are given in Section 13.

2 Nomenclature

Assume that $A$ is a matrix of size $m \times n$. The sub matrix that lies in the rows of $\alpha \subseteq \{1,..,n\}$ and columns of $\beta \subseteq \{1,..,m\}$ is denoted $A(\alpha, \beta)$. For the argument $B = \{\alpha, \beta\}$, $A(B)$ is interpreted as $A(B) = A(\alpha, \beta)$. For example

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}, \quad \alpha = \{1,2\}, \quad \beta = \{1,3\}, \quad B = \{\alpha, \beta\}$$

$$A(B) = A(\{\alpha, \beta\}) = A(\{1,2\}, \{1,3\}) = \begin{bmatrix} a_{11} & a_{13} \\ a_{21} & a_{23} \end{bmatrix}.$$ 

Further, $1:n \triangleq \{1,2,...,n\}$ and $x_{1:n} = \{x_1, x_2, ..., x_n\}$.

3 Recursive Bayesian Estimation

Consider a dynamic system with the state vector $x_n$ and measurements $y_n$ at time sample $n$

$$x_n = f(x_{n-1}, q_n) \quad (1)$$
$$y_n = g(x_n, v_n) \quad (2)$$
where \( f(\cdot) \) and \( g(\cdot) \) are possibly nonlinear vector-valued functions, \( \mathbf{q}_n \) and \( \mathbf{v}_n \) are uncorrelated white noise sequences with known statistical distributions \( p(\mathbf{q}_n) \) and \( p(\mathbf{v}_n) \).

The estimation problem in hand is to assign some degree of belief in the state \( \mathbf{x}_n \) at time sample \( n \), given the measurements

\[
\mathbf{y}_{1:n} = \{ \mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_n \}.
\]

To this end, it is required to construct the pdf (probability density function) \( p(\mathbf{x}_n | \mathbf{y}_{1:n}) \). This can be done recursively in two steps: prediction and update. Assume that at step \( n - 1 \) the pdf \( p(\mathbf{x}_{n-1} | \mathbf{y}_{1:n-1}) \) is known. Then, according to Chapman-Kolmogorov equation, the pdf \( p(\mathbf{x}_n | \mathbf{y}_{1:n-1}) \) is obtained by

\[
p(\mathbf{x}_n | \mathbf{y}_{1:n-1}) = \int p(\mathbf{x}_n | \mathbf{x}_{n-1})p(\mathbf{x}_{n-1} | \mathbf{y}_{1:n-1})d\mathbf{x}_{n-1}. \tag{3}
\]

This is the prediction step where the pdf \( p(\mathbf{x}_n | \mathbf{y}_{1:n}) \) is predicted from \( p(\mathbf{x}_{n-1} | \mathbf{y}_{1:n-1}) \) known from the previous step and \( p(\mathbf{x}_n | \mathbf{x}_{n-1}) \) given indirectly by (1). In (3), the fact that (1) describes a Markov process of order 1 is taken advantage of, which implies \( p(\mathbf{x}_n | \mathbf{x}_{n-1}, \mathbf{y}_{1:n-1}) = p(\mathbf{x}_n | \mathbf{x}_{n-1}) \). In the update step, the measurement \( \mathbf{y}_n \) is used to update the pdf with Bayes’ rule

\[
p(\mathbf{x}_n | \mathbf{y}_{1:n}) = \frac{p(\mathbf{y}_n | \mathbf{x}_n)p(\mathbf{x}_n | \mathbf{y}_{1:n-1})}{p(\mathbf{y}_n | \mathbf{y}_{1:n-1})} \tag{4}
\]

where the normalizing factor \( p(\mathbf{y}_n | \mathbf{y}_{1:n-1}) \) is given by

\[
p(\mathbf{y}_n | \mathbf{y}_{1:n-1}) = \int p(\mathbf{y}_n | \mathbf{x}_n)p(\mathbf{x}_n | \mathbf{y}_{1:n-1})d\mathbf{x}_n. \tag{5}
\]

In equations (4) and (5), \( p(\mathbf{y}_n | \mathbf{x}_n) \) is obtained from measurement equation (2) while \( p(\mathbf{x}_n | \mathbf{y}_{1:n-1}) \) is given by (3). The integrals involved in (3) and (4) are generally analytically intractable and numerical solutions would typically be too computationally demanding to evaluate.

### 4 Computer memory architecture

#### 4.1 Single-core memory architecture

A typical computer has a Random Access Memory (RAM), often referred to as the working memory, and also a smaller cache memory with faster access time in between the RAM and CPU, as shown in Fig. 1. Every data element that is requested from the CPU will be brought into the cache memory, if not already there. The idea of introducing an intermediate memory between the RAM and CPU is based on the observation of data temporal and spatial locality.
Temporal locality means that a data element that is being accessed now is likely to be referenced soon again. If the referenced data element is stored in the cache memory, it will be accessed fast when it is needed the next time, provided repeated accesses occur soon enough so that the element has not been replaced.

Spatial locality means that when a data element is accessed, it is likely that data at neighboring addresses will be accessed soon, for instance in an array that is processed in a for-loop. Therefore, if a single data element is accessed, neighboring data will also be fetched into the cache memory. The number of elements that are fetched at a time depends on the cache line size.

If an algorithm utilizes data intensively but for just a few simple calculations, i.e. the memory access to floating point operation (FLOP) ratio is high, it is crucial to handle data distribution over the memory levels in an efficient way. Fetching a data element resident in the RAM memory typically takes a few hundred CPU cycles, while doing the same but with the cache memory or performing a FLOP can be completed in a few cycles. To keep the CPU from wasting time on waiting for data to work on, it is important that data elements that are brought into the cache will be reused to complete as many calculations as possible in the algorithm before they are thrown out.

The structure in Fig. 1 with a single cache memory is a simplified description of the memory system architecture of today’s computers. Often there is a hierarchy of cache memories in between the RAM and CPU. Three levels, L3, L2, and L1, are not uncommon and usually organized so that L3 has the largest size but the longest access time, and L1 is the smallest but the fastest. In a system with L1 and L2 cache, typical values would be that L1 has a size of a few tens of kB (kilo byte) and an access time of a few cycles, while L2 cache has a size of a few MB (mega byte) and an access time a few tens of CPU cycles. However, to understand the ideas presented here it suffices to think of the cache as a single memory with much faster access time than the RAM memory.
4.2 Shared memory multicore architecture

In a shared-memory multicore architecture, there are several CPUs that can access a shared RAM, but each CPU also has a private cache memory, as shown in Fig. 1. Notice that here the CPUs share a common bus to communicate to the RAM. This is however only an example and the organization of bus connections differ among different computer architectures. The simplified view depicted in Fig. 1 shows that even though the CPUs have a shared memory, it is beneficial to construct an algorithm that will keep the data locally in the private cache memory of each CPU, improving the overall performance. Also it illustrates the fact that two processors can block memory accesses for each other since only one processor at the time can access the bus.

5 Parallel implementation speedup

The speedup one can theoretically expect from a parallel implementation of an algorithm with respect to a purely sequential version of it can be analyzed as follows. Assume that a program consists of a portion \( p_1 \) that must be executed sequentially and a portion \( p|| = 1 - p_1 \) that can be run in parallel. Further assume that there will be \( M \) processors used to execute the parallel section and that the parallel execution introduces an overhead of \( c(M) \).

The execution time \( t_M \) when run on \( M \) processors is given by

\[
t_M = p_1 + p||/M + c(M) = p_1 + \frac{(1 - p_1)}{M} + c(M).
\]

The speedup \( s \) is thus equal to

\[
s(M) = \frac{t_1}{t_M} = \frac{1}{p_1 + \frac{(1-p_1)}{M} + c(M)}.
\]

The term \( c(M) \) is somewhat difficult to estimate and it accounts for such effects as amount of communication, memory bandwidth, number of synchronization points etc. and highly depends on the architecture on which the program is executed. To get a well-performing parallel implementation there are three following main issues to consider.

**Sequential execution:**

From (6), it can be seen that a program containing a sequential portion can never achieve a greater speed up than \( s(\infty) = \frac{1}{p_1} \) (assuming \( c(M) = 0 \)). This is known as Amdahl’s law [12]. It is therefore of great importance that there are no large sequentially executed portions in a parallel implementation. For instance, a program with \( p_1 = 0.4 \), has a theoretically maximum achievable speedup limited to \( 1/0.4 = 2.5 \) times.

**Communication:** On a shared memory multicore architecture, communication among cores is accomplished through read and write accesses to the shared
memory. The performance cost is due to the need of synchronization and possibly less efficient utilization of the cache memory. The cost of communication is however substantially less than for a distributed memory architecture. Also if the communication can be arranged to occur when there is anyway need for synchronization, the cost of communication itself will not necessarily be very high.

Switching between sequential and parallel execution: Transfer from sequential to parallel execution requires overhead in forking threads that are run in parallel. Transfer from parallel to sequential execution requires synchronization and communication. Both of these transfers penalize speedup and switching should thus be kept to a minimum.

To recapitulate, an algorithm that is likely to perform well on a shared memory multicore computer is thus the one that can be implemented so that the three above issues are confined. An implementation that can execute large parts in parallel in between communication and synchronization events is referred to as coarse grained, while the opposite one is referred to as fine grained.

6 Hardware and Software

Evaluation of the implementations was made for two different systems, Grad and Kalkyl, at UPPMAX ¹. These systems are clusters consisting of several interconnected nodes. However, since the present study is not concerned with distributed performance of the algorithms, only one node at a time was employed for performance evaluation at each system. A node of Grad consists of two Intel® 2.66GHz quadcore, E5430 with 12 MB cache while a node of Kalkyl consists of two quadcore Intel® Xeon 5520, Nehalem 2.26 GHz with 8MB cache. The bandwidth of the two systems was evaluated by running the STREAM ² benchmark. It was found that the memory bandwidth to the RAM was about 5.5 GB/s (giga byte per second) and 23 GB/s at a node of Grad and Kalkyl, respectively. The reason for investigating two different architectures is to see how difference in parameters such as memory bandwidth, available cache etc. will affect the performance of the algorithms.

The implementations were written in C++ and Open MP ³ was used for parallelization of the code. Compilation was performed with the Portland Group Inc. compiler and full compiler optimization was used for all compilations.

¹UPPMAX is a high performance computing center, for more information see http://www.uppmax.uu.se.
²STREAM is benchmarking software specially designed to measure the memory bandwidth for a system, for more information see http://www.cs.virginia.edu/stream/.
³Open MP is an application programming interface that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran.
7 Parallel implementation of The Kalman Filter

The Kalman filter (KF) is one of the most widely used state estimation algorithms in control applications, mostly due to its optimality under the assumptions of linear plant dynamics as well as white measurement and process noise. Under these assumptions, KF is the analytical solution to the recursive Bayesian estimation problem given by Eq. (3) and Eq. (4). Unfortunately, for $N$ states to be estimated, the complexity of the KF is $O(N^3)$ [13], which can easily become too computationally expensive for embedded platforms. For instance, in echo cancellation, one seeks to estimate in real time the coefficients of a finite impulse response (FIR) filter with thousands of taps, to model an acoustic channel [14]. In such applications, the KF exhibits superior convergence speed, tracking performance and estimation accuracy compared to e.g. Normalized Least Mean Squares algorithm and Averaged Kalman Filter Algorithm (AKFA) [15, 16]. Further, the KF outperforms the Recursive Least Squares algorithm (RLS) in tracking time-varying parameters since the underlying mathematical model for the latter assumes the estimated parameters to be constant. The KF also offers, relative to RLS with forgetting factor, the benefit of individually set time variation of states, see e.g. [17].

Parallel implementations of the KF have been suggested over the years to improve the execution time. However, many of these schemes are hardware-specific assuming such architectures as e.g. the Connection Machine [18], distributed memory machines [19] and systolic arrays [20], and can thus not be used for a multi core implementation. Other parallelization solutions suffer from the presence of sequentially executed sections that prevent significant speedup [21, 22]. Pipelined by design algorithms [23] have input-to-output latency equal or even greater than that of a sequentially executed filter, which is not acceptable in many real time applications. No publication known to the authors of this paper addresses parallelization and cache memory handling of the KF on multicore platforms.

In this section, efficient parallelization of the KF executed on a shared-memory multicore architecture is studied and exemplified by an adaptive filtering application. The parallelization is achieved by re-ordering the KF equations so that the data dependencies are broken and allow for a well-parallelized program implementation that has the potential to exhibit linear speedup in the number of used cores. Analysis of the resulting algorithm brings about an estimate of the memory bandwidth necessary for a realization of this potential on a multicore computer.

7.1 System model and the Kalman filter

Parameter estimation in systems that can be written in the regressor form

$$\theta_{t+1} = \theta_t + \epsilon_t$$
$$y_t = \varphi_T^T \theta_t + e_t$$
is considered. Here $y_t$ is the scalar measured output, $\varphi_t$ is the (known) regressor vector that depends on the data up to time $t-1$, $\theta_t$ is the time-varying vector of $N$ parameters to be estimated, $\epsilon_t$ is the process noise, $e_t$ is the measurement noise and $t$ is discrete time. This description includes any linear single output system, but also a broad class of nonlinear systems that are linear in unknown parameters. An important property of the regressor model that will be utilized further is that the regressor vector $\varphi_t$ only contains data from time $t-1$. The Kalman filter equations for estimation of $\theta_t$ (see e.g. [17]) are given by:

$$\hat{\theta}_t = \hat{\theta}_{t-1} + K_t [y_t - \varphi_t^T \hat{\theta}_{t-1}]$$

(7)

$$K_t = \frac{P_{t-1} \varphi_t}{r_t + \varphi_t^T P_{t-1} \varphi_t}$$

(8)

$$P_t = P_{t-1} - \frac{P_{t-1} \varphi_t \varphi_t^T P_{t-1}}{r_t + \varphi_t^T P_{t-1} \varphi_t} + Q_t$$

(9)

where $\hat{\theta}_t \in \mathbb{R}^N$ is the estimate of $\theta_t$, $K_t \in \mathbb{R}^N$ is the Kalman gain, $P_t \in \mathbb{R}^{N \times N}$ is the error covariance matrix, $r_t \in \mathbb{R}$ is the measurement noise variance $\text{var}(\epsilon_t)$ and $Q_t \in \mathbb{R}^{N \times N}$ is the covariance matrix of the process noise $\text{cov}(\epsilon_t)$. A priori estimates of $\theta_0$ and $P_0$ are taken as initial conditions, if available. Otherwise it is standard to use $\theta_0 = 0$ and $P_0 = \rho I$ where $\rho$ is some "large" number.

A numerically sound alternative to (7) - (9) is the square root form known as the Square Root Information Filter (SRIF) [24]. The SRIF has a similar structure and data dependencies as (7) - (9) and the ideas presented below can be straightforwardly applied to the SRIF, as well. However to keep the description free from technical details, the KF formulation in the form of (7) - (9) is treated further.

8 Implementation

In this section, computer implementation of the KF equations (7) - (9) is discussed. First a straightforward implementation will be presented and the drawbacks of it will be explained. Thereafter it will be shown how these drawbacks can be remedied by a simple reordering of the equations, allowing for a well-parallelized algorithm suitable for multicore and, possibly, for distributed systems.

8.1 Straightforward implementation

To minimize the computational redundancy in (7)-(9), the common terms $C_t \triangleq P_{t-1} \varphi_t$, $b_t \triangleq \varphi_t^T P_{t-1} \varphi_t = \varphi_t^T C_t$ and $d_t \triangleq r_t + \varphi_t^T P_{t-1} \varphi_t = r_t + b_t$ are first calculated. This results in Alg. 1. The corresponding pseudocode is provided in Alg. 2.
Algorithm 1 Straightforward implementation of Eq. (7)-(9)

- \( C_t = P_{t-1} \varphi_t \)
- \( b_t = \varphi_t^T C_t \)
- \( d_t = r_t + b_t \)
- \( P_t = P_{t-1} + C_tC_t^T/d_t + Q_t \)
- \( \hat{y}_t = \varphi_t^T \hat{\theta}_{t-1} \)
- \( \hat{\theta}_t = \hat{\theta}_{t-1} + \frac{C_t}{d_t} [y_t - \hat{y}_t] \)

Algorithm 2 Pseudocode for implementation of Alg. 1

- for \( i = 1 : N \)
  - for \( j = 1 : N \)
    - \( C_t(i) = C_t(i) + P_{t-1}(i,j) \varphi_t(j) \)
    - end
  - \( b_t = b_t + \varphi_t(i)C_t(i) \)
  - \( \hat{y}_t = \hat{y}_t + \varphi_t(i)\hat{\theta}_{t-1}(i) \)
  - end
- \( d_t = r_t + b_t \)
- for \( i = 1 : N \)
  - for \( j = 1 : N \)
    - \( P_t(i,j) = P_{t-1}(i,j) + C_t(i)C_t(j)/d_t + Q_t(i,j) \)
    - end for
  - \( \hat{\theta}_t(i) = \hat{\theta}_{t-1}(i) + \frac{C_t(i)}{d_t} [y_t - \hat{y}_t] \)
  - end for
Algorithm 3 Reorganized implementation of Alg. 1

- $d_t = r_t + b_t$
- $\hat{\theta}_t = \hat{\theta}_{t-1} + \frac{C_t}{d_t} [y_t - \hat{y}_t]$
- $P_t = P_{t-1} + C_tC_t^T/d_t + Q_t$
- $C_{t+1} = P_t \varphi_{t+1}$
- $\hat{y}_{t+1} = \varphi_{t+1}^T \hat{\theta}_t$
- $b_{t+1} = \varphi_{t+1}^T C_{t+1}$

As mentioned, such an implementation has drawbacks. Assume that $\theta_t$ is of length $N = 2000$, a not uncommon size for, say, adaptive filtering in acoustics. $P$ would then require $N^2(8B) = 32$ MB of storage (assuming double precision, 8 B per element), which is too large to fit into the cache (recall that the cache size is typically a few MB). Thus to calculate $C$ in Alg. 2, the elements of $P_{t-1}$ will be brought into the cache as they are requested. Eventually, the elements of $P_{t-1}$ that were first brought in will be substituted by the elements currently in use. When the program later arrives at the calculation of $P_t$, the elements of $P_{t-1}$ must be brought in once again. Since $P$ is of considerable size, bringing it to the cache twice leads to a substantial increase in the execution time.

8.2 Reordering of the equations for efficient memory utilization

The reordering is based on the observation that $\varphi_{t+1}$ depends only on the data from time $t$, and can thus be made available at time step $t$. This observation enables the reformulation of Alg. 1 as Alg. 3. Why such an reordering would improve the performance becomes clear from the pseudo code given in Alg. 4 where it can be seen that once an element of $P$ is brought into the memory, it will be used to accomplish all calculations it is involved in. Therefore, squeezing the $P$ matrix twice trough the memory at each iteration is no longer needed.

8.3 Utilizing the symmetry of $P$

If $P_0$ is symmetric, it can be seen from (9) that $P$ will stay symmetric through the recursions. This should be taken advantage of, since approximately half of the calculations and memory storage can be spared. $C_t(i)$ can be rewritten to be calculated from only upper triangular elements as

$$C_t(i) = \sum_{j=i}^{N} P_t(i, j) \varphi_t(j) + \sum_{j=1}^{i-1} P_t(j, i) \varphi_t(j).$$
Algorithm 4 Pseudo code of memory efficient implementation.

- $d_t = r_t + b_t$

- for $i = 1 : N$
  - $\hat{\theta}_t(i) = \hat{\theta}_{t-1}(i) + \frac{C_t(i)}{d_t}[y_t - \hat{y}_t]$
  - for $j = 1 : N$
    * $P_{t+1}(i,j) = P_t(i,j) + C_t(i)C_t(j)/d_t + Q_t(i,j)$
    * $C_{t+1}(i) = C_{t+1}(i) + P_{t+1}(i,j)\phi_{t+1}(j)$
  - end for
  - $\hat{y}_{t+1} = \hat{y}_{t+1} + \phi_{t+1}^T(i)\hat{\theta}_t(i)$
  - $b_{t+1} = b_{t+1} + \phi_{t+1}(i)C_{t+1}(i)$
- end for

An implementation making use of only the upper triangular part of $P$ can thus be obtained by changing the $j$-loop in Alg. 4 to:

- for $j = i : N$
  - $P_{t+1}(i,j) = P_t(i,j) + C_t(i)C_t(j)/d_t + Q_t(i,j)$
  - $C_{t+1}(i) = C_{t+1}(i) + P_{t+1}(i,j)\phi_{t+1}(j)$
  - $C_{t+1}(j) = C_{t+1}(j) + P_{t+1}(i,j)\phi_{t+1}(i)$
- end for

8.4 Parallel implementation

Let $M$ be the number of CPUs used for the implementation. It can be observed by examining Alg. 4 that there are no dependencies between $i$-loop iterations, except for the adding up of $\hat{y}_{t+1}$, $b_{t+1}$ and $K_{t+1}$. Such dependencies are easily broken by using a reduction. In a reduction CPU $M$ calculates the local contribution, $s_M$, of the sum that is later added up in a sequential section to give the global sum $S = \sum_{i=1}^{M} s_M$. By doing so, a parallelization can be achieved by splitting the $i$-loop in equally large chunks of size $N/M$ (assumed to be integer), and letting each CPU process one of the chunks.

For the algorithm utilizing only the upper triangular part of $P$, there is an issue of splitting the workload among the CPUs. Splitting over the $i$-index would result in an unevenly distributed workload since the $j$-loop range from $i$ to $N$. Moreover, the splitting shall preferably be done so that each CPU can hold
locally as much of the data as possible. This can be achieved by the following splitting. First map the upper diagonal elements of $P$ to a rectangular matrix $P'$ of size $N \times (N/2 + 1)$, where the mapping from an element in $P$ to element $(i,j)$ in $P'$ is given by

$$P'(i,j) = P(i, (i+j-1) \mod N), \quad 1 \leq i \leq N, \quad 1 \leq j \leq (N/2 + 1)$$

Notice that this matrix contains $N/2$ elements more than necessary. The upper triangular block of $P$ contains $N(N+1)/2$ elements and $P'$ thus have $N(N/2 + 1) - N(N+1)/2 = N/2$ elements extra. This is to avoid the use of if-statements in the implementation and hence allow for better use of the pipeline in the CPU. An example for $N = 6$ is given below. Notice that $P'$ can be said to contain only upper diagonal elements since $P(i,j) = P(j,i)$.

$$P = \begin{bmatrix}
p_{11} & p_{12} & p_{13} & p_{14} & \cdots & 
p_{22} & p_{23} & p_{24} & p_{25} & \cdots & 
p_{33} & p_{34} & p_{35} & p_{36} & \cdots 
p_{44} & \cdots & p_{44} & p_{45} & \cdots & 
p_{55} & \cdots & \cdots & p_{55} & \cdots 
p_{66} & \cdots & \cdots & \cdots & \cdots
\end{bmatrix} \rightarrow P' = \begin{bmatrix}
p_{11} & p_{12} & p_{13} & p_{14} & 
p_{22} & p_{23} & p_{24} & p_{25} & 
p_{33} & \cdots & p_{34} & p_{35} & p_{36} & 
p_{44} & p_{45} & p_{46} & \cdots & \cdots & p_{55} & p_{56} & p_{57} & \cdots & \cdots & \cdots & \cdots & \cdots & p_{66} & \cdots & \cdots & \cdots & \cdots & \cdots
\end{bmatrix}$$

The redundant elements of $P'$ are in the last half of the last column, which is equal to the first half of the last column. The same mapping is applied to $Q$ to yield $Q$.

Splitting these calculations over the $i$-index so that CPU $m$ will loop from $i_{1,m} = N_M (m - 1) + 1$ to $i_{2,m} = N_M m$ gives a parallel implementation described in Alg. 5, where superscript $(m)$ denotes a local variable to CPU $m$.

9 Analysis of Algorithm 5

9.1 Sequential and parallel work

For one iteration of Alg. 5, $2M - 1 + N(M - 1)$ FLOP’s are executed sequentially which is negligible, assuming that $N$ is of considerable magnitude, compared to the $10(N^2 + N)$ FLOP’s that are executed in parallel. Further, the computational load performed in parallel is perfectly balanced, i.e. each processor will perform an equal amount of work in the parallel section.

9.2 Communication and synchronization

The proposed algorithm exhibits a large degree of data locality. Most importantly, each CPU will only access a part of $P$, consisting of $N(N+1)/2M$ elements, implying that it can be stored locally and no parts of $P$ will have to be communicated among the CPUs.

The variables that are involved in a reduction, i.e. $C$, $\tilde{y}$ and $b$, which consists of $(N/2 + 1) + N/M + 2$ elements, have to be communicated from the parallel
Algorithm 5

- **Sequential**
  
  \[
  \hat{y}_t = \sum_{m=1}^{M} y_t^{(m)} \\
  b_t = \sum_{m=1}^{M} b_t^{(m)} \\
  C_t = \sum_{m=1}^{M} C_t^{(m)} \\
  d_t = r_t + b_t
  \]

- **CPU m (in parallel)**

  - for \(i = i_{1m} : i_{2m}\)
    
    \[
    \hat{\theta}_t(i) = \hat{\theta}_{t-1}(i) + C_t(i)/d_t[y_t - \hat{y}_t] \\
    \text{for } j = \lfloor (N/2 + 1 - \frac{2i}{N}) \rfloor \text{ mod } N \\
    \quad \cdot k = (i + j) \text{ mod } N \\
    \quad \cdot P'_{t+1}(i, j) = P'_t(i, j) + C_t^{(m)}(i)C_t^{(m)}(k)/d_t + Q'_t(i, j) \\
    \quad \cdot C_{t+1}(i) = C_{t+1}^{(m)}(i) + P'_{t+1}(i, j) \varphi_{t+1}(k) \\
    \quad \cdot C_{t+1}(k) = C_{t+1}^{(m)}(k) + P'_{t+1}(i, j) \varphi_{t+1}(i) \\
    \]
    
    \[
    \hat{y}_{t+1}^{(m)} = \hat{y}_{t+1}^{(m)} + \varphi_{t+1}^{T}(i)\hat{\theta}_t(i) \\
    \hat{b}_{t+1}^{(m)} = \hat{b}_{t+1}^{(m)} + \varphi_{t+1}(i)C_{t+1}(i)
    \]
  
  - end for
to the sequential section. In the worst case scenario \((M = 2)\) this becomes 
\[(N/2 + 1) + N/2 + 2 = N + 3\] 
elements. Since double precision is assumed 
\((8\ \text{B per element})\), this means that for \(N = 2000\), 
\((8\ \text{B})(2000 + 3) \approx 16\ \text{kB}\) 
will need to be communicated, certainly not a large amount. The data to be 
communicated from the sequential to the parallel section are \(C, \hat{y}, b\) and the 
additional values of \(\varphi_{t+1}\).

Synchronization is required at the end of each iteration. The overhead inflicted 
by this event is independent of \(N\) and depends only on the number of CPUs 
used; the more processors are involved, the more expensive the synchronization 
is. However, the relative cost of synchronization becomes less for larger \(N\) and 
the synchronization overhead has smaller influence on the overall execution 
time.

### 9.3 Memory bandwidth

The memory bandwidth needed by the algorithm to perform \(n_{\text{iter}}\) iterations in 
\(t_{\text{tot}}\) seconds can be estimated as follows. The only data structures of consider-
able size in the algorithm are \(P\) and \(Q\). Studying how these are transferred from 
the RAM to the CPU gives a good estimate of the required memory bandwidth. 
If the matrices \(P\) and \(Q\) have a size of \(s(P)\) and \(s(Q)\) bytes respectively, transferring them from the RAM to the CPUs at each iteration requires a memory 
bandwidth of

\[
B = \frac{[s(P) + s(Q)] \cdot n_{\text{iter}}}{t_{\text{tot}}} \quad (10)
\]

Even though \(Qt\) is a matrix of size \(N \times N\), it is very often selected to be diagonal 
or sparse. This means that in most practical cases the required bandwidth 
needed is about half of that stated by Eq. 10.

As for any other parallel algorithm, one could thus not expect the above algo-
rithm to scale well for a too large or too small problem size \(N\). For small \(N\), 
the parallel overhead will become a bottleneck while for large \(N\) the available 
memory bandwidth might strangle the performance.

### 9.4 Cache miss handling

In a cache-based system, it is of outermost importance to avoid cache misses to 
get good performance. One of the main points in the reorganization yielding 
Alg. 3 is to minimize the cache misses for \(P\). Because of the reorganization 
the optimal strategy for minimizing the cache misses becomes simple. For the 
matrix \(P\), each element will only be used once in each loop iteration. There is 
thus no reason to store any of it in the cache. The remaining variables claim 
a negligible space of \((3N + 3) \cdot 8\ \text{B}\). Since they are reused several times in one 
iteration, they should be stored in the cache. For instance, with \(N = 8000\), 
which number is considered to be a large \(N\), they will require 190 kB of storage.
Table 1: Execution times in sec. for 50 iterations of 2, Alg. 4 and Alg. 5, executed on a single core on Grad and Kalkyl.

<table>
<thead>
<tr>
<th>N</th>
<th>Grad</th>
<th>Alg. 2</th>
<th>Alg. 4</th>
<th>Alg. 5</th>
<th>Kalkyl</th>
<th>Alg. 2</th>
<th>Alg. 4</th>
<th>Alg. 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>0.12</td>
<td>0.063</td>
<td>0.021</td>
<td>0.12</td>
<td>0.051</td>
<td>0.028</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>0.22</td>
<td>0.11</td>
<td>0.073</td>
<td>0.20</td>
<td>0.11</td>
<td>0.089</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>1.06</td>
<td>0.60</td>
<td>0.33</td>
<td>0.99</td>
<td>0.56</td>
<td>0.34</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4000</td>
<td>4.42</td>
<td>2.49</td>
<td>1.37</td>
<td>3.92</td>
<td>2.08</td>
<td>1.31</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8000</td>
<td>17.55</td>
<td>9.60</td>
<td>5.51</td>
<td>16.52</td>
<td>8.45</td>
<td>5.54</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This is in the order of 0.1% of a cache of a few MB. The strategy is thus to store everything except $P$ in the cache, unless all data fits in the cache completely. In the latter case all the data should certainly be kept in the cache.

10 Results

All calculations were carried out using double precision. The test data came from a simulation and were the same for all runs. Program compilation was performed with the pgi-compiler and full compiler optimization was used for all the algorithms. Open MP [25] was used for parallelization. This allowed the program to be executed in parallel by adding a single extra code line telling the compiler to run the outer $i$-loop in parallel and perform the required reductions. The matrix $Q$ was diagonal. To evaluate the improvement gained by reorganizing the equations, Alg. 2 was compared to Alg. 4. The rest of the experiments were devoted to the algorithm of main interest, i.e. Alg. 5. Also, as mentioned before, the memory bandwidth of Kalkyl and Grad were evaluated, to enable further analysis.

10.1 Execution time and speedup

Table 1 shows execution times for the memory efficient algorithm, Alg. 4, the memory inefficient algorithm, Alg. 2, and the parallelizable implementaiton Alg. 5, tested on Grad and Kalkyl. Speedup curves for Alg. 5 are plotted in Fig. 2.

10.2 Memory Bandwidth

Tab. 2 show estimates of the required memory bandwidth $B_{lin}(N,M)$ to achieve linear speedup for problem size $N$ using $M$ processors. These values were obtained by applying (10) to the data in Tab.1, to calculate $B_{lin}(N,1)$ with further extrapolation for $M \geq 1$, i.e $B_{lin}(N,M) = M \cdot B_{lin}(N,1)$. 

16
Figure 2: Speedup for Alg. 5, executed on Grad (left) and Kalkyl (right). For reference linear speedup is marked by the dashed line.

Table 2: Theoretically evaluated bandwidth to obtain linear speedup of Alg. 5 executed on Grad and Kalkyl in GB/s.

<table>
<thead>
<tr>
<th>M \ N</th>
<th>500</th>
<th>1000</th>
<th>2000</th>
<th>4000</th>
<th>8000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grad</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2.4095</td>
<td>2.7255</td>
<td>2.4038</td>
<td>2.3229</td>
<td>2.3207</td>
</tr>
<tr>
<td>2</td>
<td>4.8190</td>
<td>5.4509</td>
<td>4.8075</td>
<td>4.6458</td>
<td>4.6414</td>
</tr>
<tr>
<td>Kalkyl</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1.7585</td>
<td>2.2470</td>
<td>2.3512</td>
<td>2.2636</td>
<td>2.3086</td>
</tr>
<tr>
<td>2</td>
<td>3.5169</td>
<td>4.4940</td>
<td>4.7023</td>
<td>4.5271</td>
<td>4.6173</td>
</tr>
<tr>
<td>8</td>
<td>14.0677</td>
<td>17.9761</td>
<td>18.8093</td>
<td>18.1085</td>
<td>18.4690</td>
</tr>
</tbody>
</table>
11 Discussion

It can be seen from Table 1 that the memory-efficient algorithm, Alg. 4, executes about twice as fast as the memory-inefficient algorithm, Alg. 2, on both systems (Grad and Kalkyl). Comparing execution times for Alg. 4 and Alg. 5 in Tab 1, it can also be concluded that the execution time for the algorithm utilizing the symmetry of \( P \) runs, as expected, about twice as fast as the algorithm using the whole \( P \) matrix.

**Speedup curve for Kalkyl**

Since linear speedup is obtained for all values of \( N \), there is apparently neither problem with synchronization overhead for small values of \( N \) nor memory bus saturation for larger values of \( N \). This is further confirmed by Table 2 where none of the elements exceed the available bandwidth of 23 GB/s. Even superlinear speedup for small values of \( N \) can be observed. This is due to good cache performance. With the work distributed among several cores, each core needs to access a smaller amount of data that will fit easier into the cache and result in a better overall throughput. For a more extensive explanation of this phenomenon, see e.g. [26].

**Speed-up curve for Grad**

In the speedup curve for Grad, bad scaling for \( N = 500 \) and \( N = 1000 \) is observed. This is due to the synchronization overhead that constitutes a disproportionately large part of the execution time. Also in Tab. 2, there are indications that the memory bus would be saturated for \( N = \{500, 1000, 2000\} \) and \( M = \{4, 8\} \) since the available bandwidth of 5.5 GB/s would be exceeded for these entries. However, no saturation can be seen in the speedup curves and almost linear speedup is obtained for \( N = 2000 \). One possible explanation to this discrepancy is that the analysis in Section 9.3 assumes that \( P \) is transferred from the RAM to the CPU at each iteration. For \( N \leq 2000 \), the size of \( P \) satisfies \( s(P) \leq 16 \) MB. Since there are 24 MB cache available running on 8 cores, the whole \( P \) matrix will remain in the cache memory between iterations, avoiding the need of fetching it from the RAM, creating an illusion of a larger memory bandwidth. For \( N \geq 4000 \), \( s(P) \geq 64 \) MB, which is larger than the available cache of 24 MB, the whole matrix must be brought to the cache from the RAM at every iteration. At this point, the memory bandwidth really becomes a bottleneck. Indeed, the entries in Tab. 2 corresponding to \( N = \{4000, 8000\} \) and \( M = \{4, 8\} \) do not align with the linear speedup for \( N \geq 4000 \). Therefore, on this hardware and using the proposed KF algorithm, more bandwidth than the available 5.5 GB/s is needed to achieve a linear speedup.
12 Parallel implementation of the particle filter

The PF solves the recursive Bayesian estimation problem approximately with Monte Carlo simulation and provides a general framework for nonlinear/non-Gaussian dynamic systems. Navigation, positioning, tracking, communication, economics and also computer vision are some application areas where PFs have been applied and proved to perform well.

A special case in tracking applications is the so-called Bearings-Only Tracking (BOT) problem. This is a scenario often occurring in defense applications where only the normalized angle to a maneuvering target, relative some given reference angle, is measured. The BOT problem is inherently nonlinear with observability issues and is typically solved with respect to a set of constraints representing e.g. a geographical map. This is an application where the particle filter is intensively used.

A well-known drawback of the PF is that good estimation accuracy requires a large number, often thousands, of particles which makes the algorithm computationally expensive. Further, in tracking applications, the particle filter often constitutes only a part of a complete tracking system containing interacting multiple model and joint probabilistic data association algorithms, communication of measurements, constraint handling etc. Time limits in such systems are often tight and it is desirable to optimize each part in terms of execution time.

Since the PF algorithm is to a large extent parallel, it is natural to turn to parallel implementations to improve execution times so that real-time feasibility is achieved. It is therefore instructive and motivated to study the speedup and tracking performance of existing parallel PF algorithms implemented on a multicore architecture. By implementing a PF in parallel, a real-time feasible, powerful, energy effective and cheap filter that can handle a broad class of nonlinear dynamic systems with constraints is obtained.

In this section, four different existing parallel PFs, namely global distributed PF (GDPF) [8], resampling with non-proportional allocation filter (RNA) [9], resampling with proportional allocation (RPA) [9] and the Gaussian PF (GPF) [10] are compared in tracking accuracy and speedup at solving a testbed BOT problem. The filters are implemented on a shared memory multicore computer, using up to eight cores.

12.1 The particle filter

The idea of PF is to recursively obtain a weighted sample from $p(x_n|y_{1:n})$ by Monte Carlo simulation and evaluate an estimate $\hat{x}_n$ of $x_n$ from it. In this paper the special case when the proposal distribution is given by $p(x_n|x_{n-1})$ will be used, which is the most commonly encountered case in practice. However the investigated algorithms can be straightforwardly applied to the general particle filter as well. For the general PF algorithm see for instance [2, 3, 4]. Assume
that at time sample \( n - 1 \) the particle set \( S_{n-1} = \{x_{n-1}^{(i)}, w_{n-1}^{(i)}\}_{i=1}^{N} \) constitutes a weighted sample from \( p(x_{n-1}|y_{1:n-1}) \), where \( x_{n-1}^{(i)} \) is the \( i \)-th particle with associated weight \( w_{n-1}^{(i)} \). Given \( S_{n-1} \), a sample from \( p(x_{n}|y_{1:n}) \) is obtained by propagating each particle through system equation (1), i.e.

\[
x_{n}^{(i)} = f(x_{n-1}^{(i)}, q_{n}^{(i)})
\]  

(11)

where \( q_{n}^{(i)} \) is a draw from \( p(q_{n}) \). This corresponds to the prediction step in recursive Bayesian estimation. The measurement \( y_{n} \) is then used to update the weights by

\[
w_{n}^{(i)} = w_{n-1}^{(i)}p(y_{n}|x_{n}^{(i)}).
\]  

(12)

which corresponds to the update step in recursive Bayesian estimation. The two steps above yield the particle set \( S_{n} = \{x_{n}^{(i)}, w_{n}^{(i)}\}_{i=1}^{N} \) at time sample \( n \). By iterating (11) and (12), samples from \( p(x_{n}|y_{n}) \) are thus recursively obtained and can be used to produce an estimate \( \hat{x}_{n} \) of the state \( x_{n} \) using

\[
\hat{x}_{n} = E(x_{n}) \approx \sum_{i=1}^{N} \tilde{w}_{n}^{(i)}x_{n}^{(i)}
\]  

(13)

where \( \tilde{w}_{n}^{(i)} = w_{n}^{(i)}/\sum_{i=1}^{N} w_{n}^{(i)} \), is the \( i \)-th normalized weight. The recursion is initialized by making \( N \) draws from an a priori distribution \( p(x_{0}) \).

Resampling is used to avoid degeneracy of the algorithm. In resampling, a new set of \( N_{a} \) particles \( S_{n}' = \{x_{n}^{(i)}, w_{n}^{(i)}\}_{i=1}^{N_{a}} \), is to be created and replace the old set \( S_{n} = \{x_{n}^{(i)}, w_{n}^{(i)}\}_{i=1}^{N_{b}} \) of \( N_{b} \) particles. Usually, but not necessarily, \( N_{a} = N_{b} \). Most resampling algorithms obtain the resampled set by drawing with replacement \( N_{a} \) samples from the set \( S_{n} \) so that \( \Pr(x_{n}^{(i)} = x_{n}^{(i)}') = w_{n}^{(i)} \), where \( \Pr(\cdot) \) stands for probability. After resampling, the particles can be considered independent identically distributed samples from \( p(x_{n}|y_{n}) \). Therefore, the weights are reset, i.e. \( w^{(i)} = 1/N, i = 1, \ldots, N \). A popular resampling algorithm is Systematic Resampling (SR) [27].

The PF algorithm used in this paper is the so-called SIR (Sampling Importance Resampling) algorithm given by pseudocode in Alg. 6.

### 12.1.1 Gaussian Particle Filter

Another variant of the PF is the Gaussian Particle Filter (GPF) [10]. The additional assumption made in GPF is that the posterior distribution can be approximated by a Gaussian PDF, i.e. \( p(x_{n}|y_{1:n}) \approx \mathcal{N}(x_{n}; \mu_{n}, \Sigma_{n}) \) where

\[
\mathcal{N}(x; \mu, \Sigma) = \frac{1}{(2\pi)^{k/2}|\Sigma|^{1/2}}e^{-\frac{1}{2}(x-\mu)^{T}\Sigma^{-1}(x-\mu)}
\]  

(14)
Algorithm 6 SIR algorithm.

\[ S_n = SIR[S_{n-1}, z_n] \]

- **FOR** $i = 1 : N$
  - (P) Propagate $x_n^{(i)}$ (Eq. (11)).
  - (U) Set $w_n^{(i)}$ according to Eq. (12).

- **END FOR**

- (R) Resample $S_n$ using SR.

- Output the resampled set $S_n' = \{x_n'^{(i)}, w_n'^{(i)}\}_{i=1}^{N_a}$

is the $k$-dimensional normal distribution PDF for the random variable $x$, with mean $\mu$ and covariance $\Sigma$. The advantage gained is a simpler resampling scheme, and that only the estimated mean $\hat{\mu}_n$ and covariance $\hat{\Sigma}_n$ have to be propagated between iterations. These properties make the algorithm highly amenable to parallel implementation. Estimates of $\mu_n$ and $\Sigma_n$ are obtained as the weighted sample mean and covariance [28] given by

\[
\hat{\mu}_n = \frac{1}{W} \sum_{i=1}^{N} w_n^{(i)} x_n^{(i)} \quad (15)
\]

\[
\hat{\Sigma}_n = \frac{W}{W^2 - \mathcal{W}} \sum_{i=1}^{N} w_n^{(i)} (x_n^{(i)} - \hat{\mu}_n)(x_n^{(i)} - \hat{\mu}_n)^T \quad (16)
\]

where

\[
W = \sum_{i=1}^{N} w_n^{(i)}
\]

\[
\mathcal{W} = \sum_{i=1}^{N} (w_n^{(i)})^2
\]

The GPF algorithm is described by pseudocode in Alg. 7. One possible drawback of the GPF is that general constraints cannot be easily incorporated into the algorithm as is the case of SIR.

12.2 Parallel algorithms

The following notation is introduced.
Algorithm 7 GPF algorithm.

\[ [\hat{\mu}_n, \hat{\Sigma}_n] = GPF[\hat{\mu}_{n-1}, \hat{\Sigma}_{n-1}, z_n] \]

- **FOR** \( i = 1 : N \)
  - (R) Draw \( x^{(i)}_{n-1} \sim \mathcal{N}(\hat{\mu}_{n-1}, \hat{\Sigma}_{n-1}) \)
  - Perform (P) and (U) steps as in Alg. 6.
- **END FOR**

- Calculate \( \hat{\mu}_n \) and \( \hat{\Sigma}_n \) (Eq. (15) and (16)).
- Output estimated parameters \( \{\hat{\mu}_n, \hat{\Sigma}_n\} \).

---

\( M \) - Number of CPUs used.
\( S \) - Particle set \( \{x^{(i)}, w^{(i)}\}_{i=1}^N \).
\( W \) - Sum of weights \( \sum_{i=1}^N w^{(i)} \).
\( \hat{x} \) - Filter estimate.

Superscript \( m \) indicates CPU \( m \), e.g. \( N^{(m)} \) is the number of particles in the local particle set \( S^{(m)} = \{x^{(m,i)}, w^{(m,i)}\}_{i=1}^{N^{(m)}} \) at CPU \( m \). Common to all algorithms is that each CPU performs the propagation (P) and weight update (U) steps in Alg. 6 for \( N^{(m)} = N/M \) particles (\( N/M \) assumed to be integer). What differs between the algorithms then is how the resampling step (R) is handled. All described algorithms also utilize the fact that the global estimate can be calculated from the local estimates as

\[
\hat{x} = \frac{1}{W} \sum_{i=1}^N w^{(i)} x^{(i)} = \frac{1}{W} \sum_{m=1}^M W^{(m)} \hat{x}^{(m)}
\]

(17)

The description of the algorithms GDPF, RNA and RPA starts from a point where it is assumed that the CPUs have a local particle set \( S_{n-1} \) for time sample \( n - 1 \) and also have access to the measurement \( y_n \).

**12.2.1 Global Distributed Particle Filter (GDPF)**

GDPF [8] uses a straightforward way to perform resampling. Steps performed in chronological order within one iteration are as follows.

- **CPU \( m \) (in parallel)**
  - Perform (P) and (U) steps to obtain \( S^{(m)}_n \).
- **Sequential (one CPU only)**
Form $S_n = \bigcup_{m=1}^{M} S_n^{(m)}$.  
Calculate $\hat{x}_n$ and resample $S_n$.  
Redistribute the resampled particles to CPUs.

Since GDPF performs exactly the same calculations as the sequential PF, it exhibits the same accuracy. A drawback is of course a high communication demand inflicted by sending the particles back and forth between sequential and parallel sections. Furthermore, a large part of the program (resampling) has to be executed sequentially, limiting speedup possibilities.

### 12.2.2 Resampling with Non-proportional Allocation (RNA)

In RNA [9], resampling is performed in a suboptimal but parallel manner. Each CPU resamples the local set of particles $S_n^{(m)}$ with the locally normalized weights $\tilde{w}^{(m,i)} = w^{(m,i)}/W^{(m)}$. To avoid disturbing the statistics, the weights at each CPU after resampling are set so that $w^{(m,i)} = w^{(m,i)}W^{(m)}/W$.

A problem with RNA is that a CPU can starve, i.e. the local sum of weights, $W^{(m)}$, gets very small or even turns to machine zero. When starving occurs, computational resources are wasted on a particle set that provides little or no contribution to the final estimate. In [9], it is suggested that the problem can be resolved by at every iteration letting the CPUs exchange some portion $P$ of their particles. For instance the CPUs could form a ring and let CPU $m$ send $\lfloor N^{(m)} P \rfloor$ particles to CPU $m+1$, with the exception that CPU $M$ sends to CPU $1$. Steps performed in chronological order within one iteration, organized to allow for only one parallel section per iteration, are as follows.

#### • CPU $m$ (in parallel)

- Exchange some portion $P$ of the particles with neighboring CPUs.
- Set $i$-th weight to $w^{(m,i)}_{n-1} = w^{(m,i)}_{n-1}/W_{n-1}$.
- Perform (P) and (U) steps to obtain $S_n^{(m)}$.
- Calculate $\tilde{x}_n^{(m)}$ and $W_n^{(m)}$.
- Resample $S_n^{(m)}$ using the locally normalized weights $\tilde{w}^{(m,i)}_n = w^{(m,i)}_n/W_n^{(m)}$.
- Set $i$-th weight to $w^{(m,i)}_n = W_n^{(m)}$.

#### • Sequentially

- Calculate $\hat{x}_n$ (Eq. (17)).
- Calculate and distribute $W_n$ to each CPU.
12.2.3 Resampling with Proportional Allocation (RPA)

In RPA [9], the resampling is done by using an intermediate step in the resampling procedure called *inter-resampling*.

**Inter-resampling:** CPU $m$ calculates the local sum of weights $W(m)$. A CPU running sequentially takes $W(m)$ and now treats each CPU as a particle with weight $W(m)$ and uses the residual systematic resampling (RSR) algorithm [29] to produce $M$ replication factors $R(m)$, $m = 1, ..., M$, specifying how many particles CPU $m$ should possess after resampling. CPU $m$ will thus produce a number of particles proportional to $W(m)$. $R(m)$ is communicated to CPU $m$ which now performs *intra-resampling*.

**Intra-sampling:** At each CPU, the local particle set $S(m)$ is resampled with $N(m)$ input particles and $R(m)$ output particles using systematic resampling.

After this step, it is likely that the particles are unequally distributed among the CPU. Therefore *load balancing* is performed. CPUs with surplus particles send their excess particles to a CPU running sequentially, which distributes them to the CPUs with lack of particles. The number of particles that should be sent/received by CPU $m$ is given by $D(m) \triangleq R(m) - N(m)$. Steps performed at one iteration in chronological order for RPA are given below.

- **CPU m (in parallel)**
  - Perform (P) and (U) steps to obtain $S_n(m)$.
  - Calculate $W_n(m)$ and $\hat{x}_n(m)$.

- **Sequentially (intra-resampling)**
  - Calculate $\hat{x}_n$ (Eq. (17)).
  - Compute replication factors $R(m)$ using RSR.

- **CPU m (in parallel, inter-resampling)**
  - Resample using SR with $N(m)$ input particles and $R(m)$ output particles.
  - Calculate $D(m)$.

- **Sequentially**
  - Use $D(m)$ to distribute particles equally among CPUs.

A drawback of RPA is the unpredictability in execution time caused by the possibly unevenly distributed workload among the CPU in the inter-resampling step, where the execution time tied to the slowest CPU. The fact that there are two parallel sections with intermediate sequential sections per iteration also requires extra overhead that diminishes the speedup potential of the algorithm.
12.2.4 Gaussian Particle Filter (GPF)

The GPF is highly amenable to parallel implementation since it avoids the sequential resampling required by SIR. To simplify the notation in the description of the parallel implementation, the following variables are defined

\[
\mathcal{W}(m) \triangleq \sum_{i=1}^{N(m)} (w(m,i))^2
\]

\[
\alpha \triangleq \frac{W}{W^2 - \mathcal{W}}
\]

\[
\sigma(m) \triangleq \sum_{i=1}^{N(m)} w(m,i) x(m,i) (x(m,i))^T.
\]

The estimated mean \( \hat{\mu}(\mu_{n-1}) \) and covariance \( \hat{\Sigma}(\Sigma_{n-1}) \) obtained at time \( n-1 \) by the sequentially executed section. CPU \( m \) creates a local set of particles \( S_{n-1}(m) \) by making \( N(m) \) draws from \( N(\hat{\mu}_{n-1}, \hat{\Sigma}_{n-1}) \), and setting \( w(m,i) = 1/N(m) \), \( i = 1, \ldots, N(m) \). Each CPU then performs the (P) and (U) steps for \( S_{n-1}(m) \) to obtain \( S_{n}(m) \). From \( S_{n}(m) \) the quantities \( \{ \hat{\mu}(m), \sigma(m), W_n(m), \mathcal{W}(m) \} \) are calculated, where \( \hat{\mu}(m) = \hat{x}(m) \). A CPU running sequentially forms the estimated mean \( \hat{\mu}_n \) via (17) using the fact that \( \hat{\mu}_n = \hat{x}_n \). An estimate of \( \Sigma \) is obtained by exploiting that (16) could be rewritten as

\[
\hat{\Sigma} = \alpha \sum_{i=1}^{N} w(i) (x(i) - \hat{\mu})(x(i) - \hat{\mu})^T
\]

\[
= \alpha \sum_{i=1}^{N} [w(i)x(i)(x(i))^T - w(i)x(i)\hat{\mu}^T - \hat{\mu}(w(i)x(i))^T + w(i)\hat{\mu}\hat{\mu}^T] = \alpha[(\sum_{m=1}^{M} \sigma(m)) - W \hat{\mu}\hat{\mu}^T]
\]

where the relationship

\[
\sum_{i=1}^{N} w(i)x(i) = W \frac{1}{W} \sum_{i=1}^{N} w(i)x(i) = W \hat{\mu}
\]

is used in the third equality. Note that the final expression in (18) only makes use of the data contained in \( \{ \hat{\mu}(m), \sigma(m), W_n(m), \mathcal{W}(m) \}_{m=1}^{M} \). The following steps are performed in chronological order at timestep \( n \).

- **CPU m (in parallel)**
- Make $N^{(m)}$ draws $x_{n-1}^{(m,i)} \sim \mathcal{N}(\hat{\mu}_{n-1}, \hat{\Sigma}_{n-1})$ and set $w_{n-1}^{(m,i)} = 1/N^{(m)}$ to obtain $S_{n-1}^{(m)}$.
- Perform (P) and (U) steps to obtain $S_n^{(m)}$.
- Calculate $\{\hat{\mu}_n^{(m)}, \sigma_n^{(m)}, W_n^{(m)}, W_n^{(m)}\}$ from $S_n^{(m)}$.

- **Sequentially**
  - Use the obtained data to calculate $\hat{\mu}_n$ and $\hat{\Sigma}_n$ using Eq. (17) and Eq. (18).

### 12.3 Performance evaluation

The filters were implemented in C++ on a shared memory multicore computer, using OpenMP [25] for parallelization. Tracking accuracy was evaluated for a bearings-only tracking (BOT) application, where only the bearings, i.e. the normalized angles to the target, relative some given reference angle, were measured. This is a scenario encountered in e.g. defense applications where passive microphone array sensors are used for measurements. As a performance measure, the position RMSE, taken over 200 independent simulation runs, was studied. Note that GDPF performs the exact same calculation as the sequential PF and can thus be taken as a performance measure for the sequential PF.

**Evaluation scenario** A instant image from a representative simulation run is shown in Fig. 3. The PF is tracking a target traveling along a road, the start and stop points are marked in the figure. Two sensors are taking noisy bearing measurements of the target. The simulated measurement for sensor $i$ at time sample $n$ was obtained as the true bearing $\theta_{i,n}$ corrupted by noise, i.e. $z_{i,n} = \theta_{i,n} + q_{i,n}$ where, $q_{i,n}$ is zero mean white noise with standard deviation $\sigma_q = 0.1$ rad.

**State space model** The same state space model as in [30] was used. The system equation is given by

$$
\mathbf{x}_n = \begin{bmatrix} \mathbf{I}_2 & \mathbf{I}_2 \mathbf{T} \\ \mathbf{0} & \mathbf{I}_2 \end{bmatrix} \mathbf{x}_{n-1} + \begin{bmatrix} \mathbf{I}_2 \mathbf{T}^2/2 \\ \mathbf{I}_2 \mathbf{T} \end{bmatrix} \mathbf{w}_n
$$

where the state vector $\mathbf{x}_n = [x_n, \ y_n, \ \dot{x}_n, \ \dot{y}_n]^T$ consists of the Cartesian position and velocity components. $\mathbf{I}_2$ denotes the $2 \times 2$ identity matrix. The system noise $\mathbf{w}_n$ is white with distribution $\mathcal{N}(\mathbf{0}, \sigma_w^2 \mathbf{I}_2)$ and $T$ is the sampling period. A single measurement taken by sensor $i$ is related to the state vector by
\[ z(i,n) = g(i)(x_n) + v(i,n) \]

where \( g(i)(\cdot) \) is the trigonometric function relating the \( x\)-\( y \) position to the bearing, i.e. \( \tan^{-1}(y/x) \), if the target is in the first or fourth quadrant, considering the position of sensor \( i \) as the origin of the coordinate system, and \( v(i,n) \) is zero mean white Gaussian noise with variance \( \sigma_v^2 \). In the simulation \( \sigma_w = 10\text{m/s}^2 \), \( \sigma_v = 0.1\text{rad} \) and \( T = 0.5\text{s} \) were used.

**Performance** Fig. 4 and Fig. 5 show the tracking performance using \( M = 4 \) and \( M = 8 \) respectively. Fig. 6 shows the achieved speedup, in the figure \( P = 0.2 \) was used for RNA.
Figure 5: RMSE as a function of the total number of particles $N$. RNA-$X\%$ denotes RNA with $X\%$ particle exchange, $P$. Note the log-log scale.

Figure 6: Speed up for $N$ equals 100, 500, 1000 and 10000 particles. For reference linear speed up is marked by the dashed line.
12.4 Discussion

As can be seen from Fig. 4, GPF provides better tracking accuracy than other filters at the given scenario, especially for a small number of particles. It must though be noted that in the given scenario the measurement noise is Gaussian distributed which provides an almost ideal situation for the GPF. However, this Gaussian noise model is probably not unrealistic in ground target BOT with fixed sensor platforms. RNA with 0% particle exchange provides significantly lower tracking accuracy than other filters since it suffers from CPU starvation. All other filters have comparably the same performance as the sequential SIR filter.

For the RNA algorithm, the tracking accuracy is affected by the amount of particle exchange $P$. For smaller $N(m)$, $S_{n}^{(m)}$ gives a less accurate local approximation to $p(x_{n}|z_{1:n})$ and stronger coupling between local particle sets (larger $P$) is required to maintain tracking accuracy. This effect can be seen comparing RNA 10% and RNA 50% in Fig. 5, where for a small number of particles ($N = 100$) RNA 50% provides better tracking accuracy that RNA 10%. This effect cannot be clearly seen in Fig. 4 since the number of CPUs is less, implying larger local particle set, and less coupling is thus required to maintain tracking accuracy.

The obtained speedup naturally depends on the number of particles used, as can be seen from Fig. 6. For a small number of particles, the parallelization becomes too fine-grained, and the benefit of using a parallel implementation diminishes.

As expected, the speedup of GDPF is quite limited, restricted to about 3 times, depending on the large amount of work (resampling) that is carried out sequentially. RNA achieves speedups very close to linear for large particle sets ($N = 10^4$). The speedup of RPA is substantially less than RNA and GPF, mainly depending on the overhead caused by the two parallel sections per iteration. GPF provides the best speedup, almost linear in the number of cores used for large particle sets ($N = 10^4$).

13 Conclusions

This article investigates how two of the most powerful tools for state estimation in control and signal processing, the Kalman filter and the particle filter, can be efficiently implemented in parallel on a multicore platform. It is concluded through test runs on two different multicore architectures that it is indeed possible to achieve linear speed up in the number of cores used. Conclusions specific to each of the filters are summarized below.
13.1 Conclusions for parallel implementation of the Kalman filter

Through test runs on two different shared-memory multicore architectures, it is found that a Kalman filter for adaptive filtering can be efficiently implemented in parallel by organizing the calculations so that the data dependencies are broken. The proposed algorithm executes about twice as fast on a single core as a straightforward implementation and is capable of achieving linear speedup in the number of cores used. However, since the KF involves relatively simple calculations on large data structures, it is required that the hardware provides enough memory bandwidth to achieve linear speedup. This is an inherent problem of the KF itself and not caused by the proposed parallelization algorithm.

13.2 Conclusions for parallel implementation of the particle filter

Simulations performed with four different parallel PF algorithms showed that in a BOT problem the GPF gave best tracking performance while GDPF, RNA and RPA demonstrated tracking performance comparable to that of the sequential SIR algorithm. The drawback of the GPF is that it requires the posterior distribution to be approximately normally distributed, which is probably not unrealistic in ground target BOT with fixed sensor platforms.

The obtained speedups gained on a shared memory multicore computer depend largely on the total number of used particles $N$. For particle sets with $N > 1000$ GPF and RNA can achieve close to linear speedups in the number of cores used. The speedup obtained by RPA is substantially lower due to less beneficial parallelization potential. GDPF has a speedup limited to about 3.5 times as a consequence of the sequentially executed part of the algorithm. For particle sets with $N < 500$, the parallelization becomes too fine-grained, and it is hard to exceed a speedup of about 2 times using GDPF or RPA while GPF and RNA can achieve a speedup of up to about 4 times.

The final conclusion for a parallel particle filter implemented on a shared memory multicore computer is thus the following. If the Gaussian assumption made by GPF holds true, it would be the algorithm to prefer since it provides best tracking accuracy and is capable of achieving close to linear speedups in the number of cores used. If the Gaussian assumption does not hold, RNA would be the algorithm to prefer since it as well can, without loss in accuracy compared to the sequential PF, obtain close to linear speedups in the number of cores used.
References


32


Parallelization of the Kalman Filter for Banded Systems on Multicore Computational Platforms

Olov Rosén, Alexander Medvedev, Torbjörn Wigren*†

Abstract

Parallelization of the Kalman filter algorithm, with emphasis on the specific demands of multicore architecture implementation, is investigated. The approach is based on the nonrestrictive assumption of a banded system matrix. Both time-varying and time-invariant systems can be generally transformed to such a form. The proposed method is applied to a radio interference power estimation problem for which speedup evaluations using up to eight cores are performed. It is shown that the algorithm is capable of achieving linear speedup in the number of cores used, while speedup factors for a parallel BLAS implementation are less than two. An algorithm analysis that provides guidelines to the choice of implementation hardware to meet a desired performance is also provided.

1 Background

Parallelization of signal processing and control algorithms is a topic that is more important than ever. Due to physical limitations, as well as power and heat dissipation concerns, the performance of single core computers has reached the limit and stopped evolving. All major processor manufacturers have moved on to multicore designs in order to increase the computational capacity of their machines. Due to high performance, low energy consumption and heat dissipation, multicore architectures are highly appealing for computationally demanding embedded control and signal processing applications. Low-power embedded platforms in e.g. communication are often based on parallel hardware because of the substantially lower power-per-FLOP ratio.

To exploit the increase in computational power of the multicore hardware, the algorithms must be modified to suit the architecture. In fact, a poorly

*The two first authors are partially supported by the project "Computationally Demanding Real-Time Applications on Multicore Platforms" funded by Swedish Foundation for Strategic Research.
†O. Rosén and A. Medvedev are with the Institution of Information Technology, Department of System and Control, Uppsala University, Uppsala SE-751 05, Sweden \{olov.rosen, alexander.medvedev\}@it.uu.se
parallelized algorithm can, in many cases, execute even slower on a parallel machine than on a sequential one with the same clock rate. This poses a major problem to companies dependent on hard real-time systems: they can actually face a slow-down of their software due to the oncoming departure of sequential processors.

The Kalman Filter (KF) still represents the mainstay of linear estimation, even in medium and large-sized systems. Parallel implementations of the KF have been suggested over the years to improve the execution time. However, many of these schemes are hardware-specific with respect to such architectures as e.g., the Connection Machine [1], distributed memory machines [2] and systolic arrays [3] and thus are not directly suitable for a multicore implementation. Other parallelization solutions suffer from the presence of sequentially executed sections that prevent significant speedup [4], [5]. Pipelined-by-design algorithms [6] have input-to-output latency equal or even greater than that of a sequentially executed filter, which property is not acceptable in many real-time applications (e.g., the application in cellular communication studied in [7]). In [8], a parallel multicore implementation of the KF for parameter estimation is presented. In the present work, the method of [8] is extended to include state estimation for time varying systems with a banded system matrix. Notably, a similar analysis for state estimation in stochastic nonlinear systems by means of Particle Filters is presented in [9].

This paper deals with efficient parallel implementation of the Kalman filter (KF) for state estimation in discrete time-varying linear systems on shared memory multicore architectures. However, the proposed solution requires only a small amount of inter-processor communication, which makes it suitable also for a distributed architecture. The KF algorithm consists of a sequence of matrix-matrix and matrix-vector multiplications. The parallelization of this kind of operations on a multicore architecture is a routine matter. It is indeed possible to apply it to the steps of the KF algorithm one-by-one in a straightforward manner. However, as it is shown in this paper, the result will suffer from several drawbacks that include a large amount of inter-processor communication, synchronization, and high demand for memory bandwidth. For the case of systems with banded state-space realizations, the above mentioned drawbacks of parallel KF implementation can be efficiently alleviated.

As the Multiple-Input Multiple-Output (MIMO) estimation problem with $p$ outputs always can be implemented as a sequence of $p$ single-output filter problems, a method for the Multiple-Input Single-Output case is developed and used as a building block for the MIMO case. This approach avoids the inversion of a $p \times p$ matrix, which is known to be difficult to parallelize efficiently because of its intricate data dependencies.

To mention a few, active noise cancellation [25], climate and weather prediction [24], and mobile broadband load estimation are applications where efforts are made to decrease the execution time of the filtering step. The method suggested in the present paper is applied to a Wideband Code Division Multiple Access (WCDMA) load estimation problem that is deemed critical in mobile
broadband. This is a field where the computational burden is growing rapidly due to the increasing number of smart phones in the system. Since the number of users is directly affecting the number of estimated states, it is clear that the computational burden of sequential implementations of the KF becomes prohibitively demanding. The multicore techniques of the present paper provide therefore an interesting alternative to an increase in the potential number of uplink users of the cell, which in turn results in savings in the required amount of hardware and power consumption of the system.

The paper structure is as follows. In Sec. 3, a brief explanation of the assumed multicore architecture is given. Sec. 4 provides a summary of the KF equations. A discussion on banded systems is given in Sec. 5. In Sec. 6 the main contribution, a parallel implementation of the KF for MISO systems, is provided. An analysis yielding estimates of the amount of parallelizable work, the required bandwidth, and amount of communication is also given, to offer instrumental guidelines for the choice of implementation hardware. In Sec. 7, the parallelization of the KF for MIMO system based on the MISO implementation is carried out. Finally, in Sec. 8, the results of computer experiments are presented followed up by a discussion in Sec. 9.

2 Notation

Matrices and vectors are denoted by upper case bold and lower case bold letters, respectively. Let $A$ denote a matrix of size $m \times n$. The submatrix that lies in the rows of $\alpha \subseteq \{1, \ldots, n\}$ and columns of $\beta \subseteq \{1, \ldots, m\}$ is denoted $A(\alpha, \beta)$. For example, if $\alpha = \{1, 2\}$, $\beta = \{1, 3\}$ and

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix},$$

then

$$A(\alpha, \beta) = A(\{1, 2\}, \{1, 3\}) = \begin{bmatrix} a_{11} & a_{13} \\ a_{21} & a_{23} \end{bmatrix}.$$
3 Multicore architecture

Fig. 1 shows a simplified model of a shared memory multicore architecture with four central processing units (CPUs). The CPUs have a private cache memory and are connected to the RAM (the shared memory) via a shared bus. The access time to the private cache is several orders of magnitude faster than the access time to the RAM. This simplified model shows that even though the CPUs have a shared memory, it is beneficial (performance wise) to construct an algorithm that will keep the data locally in the private cache memory of each CPU. It also illustrates the fact that two processors can block memory accesses for each other since only one processor at the time can access the bus. Interprocessor communication is accomplished through read and write accesses to the shared memory.

![Diagram of a shared memory multicore architecture](image)

Figure 1: Simplified model of a shared memory multicore architecture.

4 The Kalman filter

4.1 State space system description

Consider a MISO discrete time system

\[
\begin{align*}
\mathbf{x}_{k+1} &= F_k \mathbf{x}_k + G_k \mathbf{u}_k + \mathbf{w}_k \\
y_k &= h_k \mathbf{x}_k + j_k \mathbf{u}_k + v_k
\end{align*}
\]

with the state vector \( \mathbf{x}_k \in \mathbb{R}^n \), the input vector \( \mathbf{u}_k \in \mathbb{R}^m \) and the output \( y_k \in \mathbb{R} \) at discrete time step \( k \). Generally, \( F_k \in \mathbb{R}^{n \times n} \) and \( G_k \in \mathbb{R}^{n \times m} \) are time-varying matrices, while \( h_k \in \mathbb{R}^{1 \times n} \) and \( j_k \in \mathbb{R}^{1 \times m} \) are time-varying vectors. The process and measurement noise sequences \( \mathbf{w}_k \in \mathbb{R}^n \) and \( v_k \in \mathbb{R} \) are assumed to be independent, white, zero mean, Gaussian distributed, with the covariance matrices \( E[\mathbf{w}_k \mathbf{w}_k^T] = Q_k \) and \( E[v_k^2] = r_k \), respectively.
4.2 Kalman filter equations

The KF equations below are in the so-called standard form. For filtering problems that require special attention to numerical stability, the square root formulation is to prefer [10]. Parallelization of the square root form of the KF is investigated in [1], where mainly the Givens rotation step is parallelized. However, many systems do not require the square root form to maintain numerical stability. As it will be shown here, the implementation and parallelization can be made far more efficient for the KF in the standard form, than the implementation proposed in [1].

The KF consists of two steps: prediction and update. These are recursively applied to the data to calculate the state estimate \( \hat{x} \) and the error covariance matrix \( P \). For the system (1)-(2), the KF [11] is calculated as:

**Prediction**

\[
\begin{align*}
\hat{x}_{k|k-1} &= F_k \hat{x}_{k-1|k-1} + G_k u_k \\
P_{k|k-1} &= F_k P_{k-1|k-1} F_k^T + Q_k
\end{align*}
\]

**Update**

\[
\begin{align*}
\tilde{y}_k &= y_k - h_k \hat{x}_{k|k-1} - j_k u_k \\
d_k &= h_k P_{k|k-1} h_k^T + r_k \\
K_k &= P_{k|k-1} h_k^T d_k^{-1} \\
\hat{x}_{k|k} &= \hat{x}_{k|k-1} + K_k \tilde{y}_k \\
P_{k|k} &= (I - K_k h_k) P_{k|k-1}
\end{align*}
\]

5 Banded systems

5.1 Transformation to a banded system form

Any linear finite-dimensional system can, with more or less effort, be transformed to a realization with a banded system matrix in a numerically stable manner. This holds true for both time-varying and time-invariant systems. The number of bands in the system matrix is denoted \( N_b \), where e.g. a diagonal and a tridiagonal matrix have \( N_b = 0 \) and \( N_b = 1 \), respectively. Under the state variable transformation \( x_k = T_k z_k \), where \( T_k \) is a non-singular matrix, the transformed system is given by

\[
\begin{align*}
z_{k+1} &= F'_k z_k + G'_k u_k + T_{k+1}^{-1} w_k \\
y_k &= h'_k z_k + j_k u_k + v_k
\end{align*}
\]

where
5.2 Time-invariant case

Assume that (1)-(2) represent a time-invariant system, i.e. \( F_k = F, G_k = G, h_k = h \), \( \forall k \). Then the following holds:

- For \( F \) with distinct eigenvectors, a modal form can be used. A transformation \( T \) can be found such that \( F' \) is block-diagonal with a \( 2 \times 2 \) block for each complex conjugated pair of eigenvalues and a \( 1 \times 1 \) block for each real eigenvalue of \( F \) [23].

- There is also a possibility to bring the system (the \( F' \) matrix) to a tridiagonal form, in a numerically sound manner. Tri-diagonalization via similarity transforms as well as tri-diagonal realizations obtained directly from the system’s Hankel matrix are discussed in [12].

- It is always possible to transform \( F \) to Jordan (bi-diagonal) form [23]. However, since the Jordan form is known to exhibit poor numerical properties, it is not always a suitable option.

Thus, the matrix \( F' \) belongs to the class of tri-diagonal matrices \( (N_b = 1) \) in the worst case and, in the best case, it is diagonal \( (N_b = 0) \). Both cases make the KF equations highly suitable for parallel implementation. Note that, for time-invariant systems, the transformation \( T \) can be computed beforehand and offline.

5.3 Time-varying case

For time-varying systems, there are many possibilities to express the system in banded form.

- Using a realization method based on a sequence of Markov parameters, \( M_{k,j} \), it is sometimes possible to find a rank factorization such that \( M_{k,j} = C_k B_j \). A realization with a diagonal \( F \) matrix is then simply obtained by taking \( F_k = I, G_k = B_k, h_k = C_k, j_k = 0 \), [13].

- Assume that a realization is readily available in the form of (1)-(2). It can then be seen that the transformation \( T_k = \Phi_k \), where \( \Phi_k = \prod_{i=0}^{k} F_i \) is the transition matrix, will give \( F'_k = I \). Since \( G'_k \) requires the inverse transform \( T_{k+1}^{-1} = \Phi_{k+1}^{-1} \), it is not always computationally sound. However, in case analytical expressions for \( \Phi \) and \( \Phi^{-1} \) are available or can be
obtained by a small amount of computation, the transformation can be applied to obtain a diagonal system.

- If $F_k$ is a sparse matrix where the zero elements are located at the same positions for all $k$, an optimized band structure of $F'_k$ can be obtained by taking $T$ to be a permutation matrix [14].

- A system that consists of loosely coupled subsystems can often be realized with a block diagonal matrix $F$ that possesses a few block-diagonal elements describing the couplings between the subsystems.

- Matrices arising from finite-element or finite-difference problems in one or two dimensions are often banded. The bandedness stems from the fact that the variables are not coupled over arbitrarily large distances. For instance, discretizations of the differential equation

$$\frac{\partial^{n_1}T(x_1,x_2)}{\partial^{n_1}x_1} = \frac{\partial^{n_2}T(x_1,x_2)}{\partial^{n_2}x_2}$$

$0 \leq n_1, n_2 \leq 2$ encountered in physics as the heat equation, wave equation and Laplace equation, have realizations in a banded form.

- In stochastic setups, where the KF is used as a black-box parameter estimator [13], the parameter vector $\theta$ is modeled as a random walk, and the output is required to be a linear combination of the unknown parameters driven by process noise. The system equations are then given by

$$\theta_{k+1} = \theta_k + w_k$$
$$y_k = h_k\theta_k + v_k$$

which is a special case of (1)-(2) with $F_k = I$, $G_k = j_k = 0$. Since $F_k = I$, the recursive parameter estimation problem is especially suitable for parallel implementation [8].

6 MISO System

The KF problem in MISO systems is treated for the parameter estimation case ($F_k = I$) in [16]. There, it is shown that the KF can be efficiently implemented on a single core and provides enough parallelism to achieve linear speedup in the number of cores used for a multicore implementation. The presence of a general structure matrix $F$ in the system equations makes efficient implementation and parallelization of the KF somewhat more difficult. As discussed in Sec. 5, it is always possible to yield a realization, or transform an existing realization, so that the matrix $F$ becomes banded with a low band width. In such a case, it is possible to formulate an extension of the technique suggested in [16]. Since the main purpose is to achieve faster execution times, it is of importance to optimize
the implementation of the sequential version, from which the parallel version will be built. Therefore, an efficient sequential implementation is presented below in Section 6.1 and the parallelization of it is handled in Section 6.2.

6.1 Efficient sequential implementation

The main focus of optimization should be on the computations involving the matrix $P$ since a majority of the FLOPs and memory accesses are related to it. In Alg. 1, an implementation of Eq. (3)-(9) where the accesses to $P$ occur after each other is given. The gain originates from the fact that, for a banded $F$, once an element of $P$ is brought to the cache, it can be used to accomplish all calculations the element is involved in before it is thrown out. Further, it allows the calculated elements in $P_{k+1|k}$ to be stored at the same locations as the elements of $P_{k|k}$ were held, giving a substantial reduction in the memory size and bandwidth needed. In [16], this reordering is shown to execute about twice as fast as compared to an implementation that does not make use of this possibility. This kind of optimization is not possible with a dense matrix $F$.

The matrix $P$ is a symmetric positive definite matrix. This should be taken advantage of since approximately half of the computations and memory storage can be spared due to this fact. However, to avoid too many technical details, the parallelization principles of KF will be presented for a version where the whole matrix $P$ is used in the computations. The modifications needed for an implementation using only the upper triangular part of $P$ are straightforward and minor.

**Algorithm 1 Efficient Kalman Filter implementation.**

\begin{align*}
\dot{x}_{k|k} &= \dot{x}_{k|k-1} + d_{k}^{-1}c_{k}[y_{k} - \hat{y}_{k}] \quad (10) \\
P_{k|k} &= P_{k|k-1} - d_{k}^{-1}c_{k}c_{k}^{T} \quad (11) \\
P_{k+1|k} &= F_{k+1}P_{k|k}F_{k+1}^{T} + Q_{k+1} \quad (12) \\
c_{k+1} &= P_{k+1|k}h_{k+1}^{T} \quad (13) \\
\hat{x}_{k+1|k} &= F_{k+1}\hat{x}_{k|k} + G_{k+1}u_{k+1} \quad (14) \\
\hat{y}_{k+1} &= h_{k+1}\hat{x}_{k+1|k} + j_{k+1}u_{k+1} \quad (15) \\
d_{k+1} &= r_{k+1} + h_{k+1}c_{k+1} \quad (16)
\end{align*}

6.2 Parallel implementation

Assume that $F$ is a dense matrix. A parallel implementation of Alg. 1 can be produced by parallelizing each step individually using BLAS\footnote{Basic Linear Algebra Subprograms (BLAS) are routines that provide standard building blocks for performing basic vector and matrix operations. BLAS is a de facto application programming interface standard, see netlib.org/blas/.} or some other
highly optimized library for matrix operations. The calculation of (12) is then split as

\[
\mathbf{A} = \mathbf{P}_{k|k} \mathbf{F}_{k+1}^T \\
\mathbf{P}_{k+1|k} = \mathbf{F}_{k+1} \mathbf{A} + \mathbf{Q}_{k+1},
\]

where each line is parallelized separately. However, such an approach will have several drawbacks. Each processor must touch a large amount of data limiting the scalability of the implementation. A synchronization point between the calculations and a temporary storage for \( \mathbf{A} \) is also required. A large amount of inter-processor communication is needed that will have negative impact on the execution time and as well limit the algorithm performance in a distributed implementation. In the case of a banded matrix \( \mathbf{F} \), where the number of bands \( N_b \ll N \), it is possible to remedy the mentioned drawbacks and thus achieve fast execution and good scalability.

Assume \( N/M \) to be integer and define (recall that \( 1 : n = \{1,2,\ldots,n\} \))

\[
\mathbf{r}_i := \frac{N}{M} (i - 1) : \frac{N}{M} i - 1 \\
\mathbf{s}_i := \mathbf{r}_i (1) - n_b : \mathbf{r}_i (N/M) + n_b
\]

A parallelization of the KF over the whole sequence of matrix operations for a banded matrix \( \mathbf{F} \) is described in Alg. 2. The algorithm is designed to make the number of synchronization points, amount of communication, and the amount of data that each processor has to touch, as small as possible. Notice that Eq. (12) in Alg. 1 is executed by the \( i \)th CPU as

\[
\mathbf{P}_{k+1|k}(\mathbf{r}_i,:) = \mathbf{F}_{k+1}(\mathbf{r}_i,\mathbf{s}_i)\mathbf{P}_{k|k}(\mathbf{s}_i,:)\mathbf{F}_{k+1}^T.
\]

Each processor is given access to the whole matrix \( \mathbf{F} \) that is banded and contains a small amount of data, but only a restricted part of \( \mathbf{P} \) is touched. Processor \( i \) will be responsible for updating of \( \mathbf{P}_{k+1|k}(\mathbf{r}_i,:) \), which will only require knowledge of \( \mathbf{P}_{k|k}(\mathbf{s}_i,:) \). The parts of \( \mathbf{P} \) that must be communicated by processor \( i \) are thus only the \( N_b \) rows that overlap with the neighboring processors (CPU \( i - 1 \) and CPU \( i + 1 \)).
Algorithm 2 Kalman Filter parallel implementation.

- Parallel (CPU \( i \) calculates)

\[
\begin{align*}
\dot{x}_{k|k}(r_i) &= \dot{x}_{k|k-1}(r_i) + d_k^{-1}c_k(r_i)[y_k - \bar{y}_k] \\
P_{k|k}(s_i,:) &= P_{k|k-1}(s_i,:) - d_k^{-1}c_k(s_i)c_k^T \\
P_{k+1|k}(r_i,:) &= F_{k+1}(r_i,s_i)P_{k|k}(s_i,:)F_{k+1}^T + Q_{k+1}(r_i,:)
\end{align*}
\]

\[
\begin{align*}
c_{k+1}(r_i) &= P_{k+1|k}(r_i,:)h_{k+1}^T \\
\dot{x}_{k+1|k}(r_i) &= F_{k+1}(r_i,s_i)\dot{x}_{k|k}(s_i) + G_{k+1}(r_i,:)u_{k+1}
\end{align*}
\]

- Sequential

\[
\begin{align*}
\bar{y}_{k+1} &= \sum_i M \bar{y}_{k+1}^{(i)} + j_k u_k \\
d_{k+1} &= r_{k+1} + \sum_i M b_{k+1}^{(i)}
\end{align*}
\]

6.3 Analysis

An analysis of Alg. 2 is carried out in this section to evaluate the number of sequential and parallel FLOPs, the required memory bandwidth, the demand of communication, and synchronization in the implementation. This provides important guidelines for the choice of hardware to meet a desired performance of the designed system.

6.3.1 Parallelizable amount of work

Counting the number of FLOPs \( f_s \) and \( f_p \) that are executed sequentially and in parallel in Alg. 2, the following expressions can be obtained

\[
\begin{align*}
f_s(M,m) &= 3M + 2m, \quad (17) \\
f_p(N,N_b,m) &= (N_b^2 + 2N_b + 5)N^2 + 2(2 + m)N. \quad (18)
\end{align*}
\]

Amdahl’s law [17] states that the maximal theoretically obtainable speedup is given by
\[ s(M) = \frac{1}{p| + \frac{p\parallel}{M}} \]

where \( p| \) and \( p\parallel \) are the sequentially and parallelly executed portion of the program, respectively. Now, if \( N > M \), which should definitely be the case, then \( f_p \gg f_s \) and \( p| \approx 0, p\parallel \approx 1 \) is a reasonable approximation, yielding \( s(M) = M \). Thus, regarding the portion of parallelizable work, the algorithm has the potential of achieving good scalability.

### 6.3.2 Memory bandwidth

The only variables of considerable size in the KF algorithm are the matrices \( P \) and \( Q \). Let \( q(P) \) denote the size of \( P \) in bytes. Assuming that \( P \) and \( Q \) are transferred from the RAM to the processors at each iteration will give a required memory bandwidth of

\[ B = n \cdot \frac{q(Q) + q(P)}{T} \quad (19) \]

to perform \( n \) iterations in \( T \) seconds. If the bandwidth \( B_h \) provided by the hardware satisfies \( B_h \geq B \), it will not be a bottleneck in the implementation. In many practical cases, \( Q \) is diagonal, or at least sparse, in which case \( q(Q) \ll q(P) \) and \( q(Q) \) can be neglected in (19).

### 6.3.3 Synchronization and Communication

Only one synchronization point at the end of the parallel section is needed. The data that have to be communicated between the CPUs are given by the overlapping rows of \( P \), the local parts of \( c \), and the variables \( y_k^{(i)}, b_k^{(i)} \), which number of elements are given by

\[ C(N, N_b, M) = 2N(M - 1)(2N_b + 2 + \frac{1}{M}). \quad (20) \]

### 7 MIMO System

Consider (1)-(2) as a MIMO system with \( p \) outputs. Denote the process noise \( v_k = [v_k(1) \quad v_k(2) \cdots v_k(p)]^T \). If \( v_k(i) \) is independent of \( v_k(j) \) \( j \neq i \), \( i = 1, 2, ..., p \), then, by the Gaussian assumption on \( v_k \), it is equivalent to \( R_k = E[v_kv_k^T] \) being a diagonal matrix. The resulting MIMO problem can be treated as a sequence of \( p \) MISO filtering problems where the filtering of the \( p \) measurements can be done sequentially, one by one [18]. If \( R_k \) is not diagonal but positive definite, a (Cholesky) transformation \( z_k = L_ky_k \) can be applied to render \( S_k = E[z_kz_k^T] \) diagonal. From the relation

\[ E[z_kz_k^T] = L_kE[y_ky_k^T]L_k^T = L_kR_kL_k^T \]
it is seen that the choice \( \mathbf{L}_k = \mathbf{R}_k^{-1/2} \) will give \( E[\mathbf{z}_k \mathbf{z}_k^T] = \mathbf{I} \), which together with the Gaussian assumption on \( \mathbf{v}_k \) establish the independence of the measurement noise. By the assumption \( \mathbf{R}_k > 0 \), \( \mathbf{R}_k^{-1/2} \) is guaranteed to exist. When there are measurements that are linearly dependent or noise-free, \( \mathbf{R} \) becomes positive semidefinite and such measurements can be handled separately by e.g. a reduced observer. The MIMO filtering problem can thus always be split to a sequence of \( p \) MISO filtering problems and the parallelization can be performed over each MISO filtering problem as proposed in Sec. 6.

8 Implementation example

In order to quantify and validate the multicore computational gains on a realistic problem, a simulation study of a WCDMA uplink interference power estimation system was used.

8.1 Uplink interference power estimation model

In this section, a simplified model of the interference power generation and measurements in the WCDMA uplink is provided. 3G mobile broadband data traffic is based on high speed packet access (HSPA) technology. The estimation of uplink load in this system is an example where the proposed parallelized KF algorithm may find application.

In the uplink, scheduling is required to assign users to the available cells. Efficient scheduling requires the interference power from users of the own cell and from users in neighboring cells to be estimated in real time, which in practice is a very difficult problem. The reference [7] therefore proposes a new algorithm for recursive Bayesian estimation of the noise power floor. Kalman filtering for uplink interference power estimation was treated in [19]. This solution does however use at least one state per user and is computationally complex. With the hundreds of users per cell anticipated in the near future, it is clear that the solution suggested in [19] becomes practically infeasible.

8.1.1 A state space model

A brief description of the state space model for WCDMA power link estimation problem is provided in this section, see [19] and [20] for a more extensive exposition. A state space model for the system is given by:

\[
\begin{align*}
\mathbf{x}_{k+1} &= \mathbf{F}\mathbf{x}_k + \mathbf{G}\mathbf{u}_k + \mathbf{w}_k \\
\mathbf{y}_k &= \mathbf{H}_k\mathbf{x}_k + \mathbf{e}_k
\end{align*}
\]
where

\[
F = \begin{bmatrix}
1 - \kappa & 0 & \cdots & 0 \\
0 & \ddots & \ddots & \vdots \\
\vdots & \ddots & 1 - \kappa & 0 \\
0 & \cdots & 0 & 1
\end{bmatrix}
\]

\[
G = \begin{bmatrix}
\kappa & \cdots & \kappa & 0 \\
1 & \cdots & \cdots & \cdots \\
0 & 1 & \cdots & 0
\end{bmatrix}
\]

\[
H_k = \begin{bmatrix}
1 + \eta_k(1) & 0 & \cdots & 0 \\
0 & \ddots & \ddots & \vdots \\
\vdots & \ddots & 1 + \eta_k(N) & 0 \\
1 & \cdots & 1 & 1
\end{bmatrix}
\]

\[
Q = \begin{bmatrix}
q(1) & 0 & \cdots & 0 \\
0 & \ddots & \ddots & \vdots \\
\vdots & q(N) & 0 & \cdots \\
0 & \cdots & 0 & q^{n+thermal}
\end{bmatrix}
\]

\[
R = \begin{bmatrix}
r(1) & 0 & \cdots & 0 \\
0 & \ddots & \ddots & \vdots \\
\vdots & r(N) & 0 & \cdots \\
0 & \cdots & 0 & r_{RTWP}
\end{bmatrix}
\]

and

\[
x_k = \begin{bmatrix}
x_k(1) & \cdots & x_k(N) & x_k^{n+thermal}
\end{bmatrix}^T
\]

\[
u_k = \begin{bmatrix}
x_k^{ref}(1) & \cdots & x_k^{ref}(N) & 0
\end{bmatrix}^T
\]

\[
w_k = \begin{bmatrix}
w_k(1) & \cdots & w_k(N) & w_k^{n+thermal}
\end{bmatrix}^T
\]

\[
y_k = \begin{bmatrix}
y_k(1) & \cdots & x_k(N) & y_{RTWP,k}
\end{bmatrix}^T
\]

\[
e_k = \begin{bmatrix}
e_k(1) & \cdots & e_k(N) & e_{RTWP,k}
\end{bmatrix}^T.
\]

The power consumed by the \(i\)th channel is \(x_k(i)\), and \(x_k^{n+thermal}\) is the sum of neighbor cell interference and thermal noise power and modeled as a random walk

\[
x_{k+1}^{n+thermal} = x_k^{n+thermal} + w_k^{n+thermal},
\]

where \(w_k^{n+thermal}\) is the systems noise corresponding to the state and \(\kappa\) is a parameter determined by the radio link quality and set for an inner control loop.

The reference power \(x_k^{ref}(i)\) for the \(i\)th channel is controlled by an outer loop controller and is given by
\[ x_{k}^{\text{ref}}(i) = \frac{1 + \eta_k(i)}{1 + (C/I)_{k}^{\text{ref}}(i)} x_{k}^{\text{total}}, \]

where \((C/I)_{k}^{\text{ref}}(t), i = 1, ..., N\) denote the carrier to interference levels. Furthermore, \(x_{k}^{\text{total}}\) is the total power and \(\eta_k(i)\) is the quotient between the data power offset and the control signal power, see [19] for details. Note that \(\eta_k(i)\), and hence \(H_k\), is time varying.

The control signal power \(y_k(i)\) is the quantity measured for the \(i\):th radio link. The additional measurement \(y_{RTWP,k}\) available on the uplink is the total received wideband uplink power. This is simply the sum of the powers represented by all states. The measurement and process noise covariance matrices are given by \(Q\) and \(R\) respectively.

In the simulation, the SINR targets were set 5 dBs lower than usually assumed for a conventional RAKE receiver, to be able to run up to 400 radio links. This is motivated since today the uplink is used with higher block error rate than when the system was standardized. Furthermore, more advanced receivers than the RAKE are in operation today, among these interference suppressing [21] and interference canceling receivers [22].

### 8.2 Results

Alg. 2 was implemented and compared with Alg. 1 parallelized using Intel’s MKL BLAS library. The execution times for a range of problem sizes are summarized in Tab. 2. The scalability is illustrated by the speedup plots depicted in Fig. 2 and Fig. 3, respectively. To verify the correctness of the parallel implementation, the same data were filtered with a sequential Matlab implementation confirming that the outputs were identical. The sum of residuals

\[ r(N_i) = \frac{1}{N_i} \sum_{k=0}^{N_i} |y_k(i) - C_k(i)x_k(i)| \]

for one channel is given for the sequential and parallel filters in Tab. 1, where \(N_i = 1000\) is the number of iterations executed.

The code was written in C and OpenMP \(^2\) was used for parallelization. The hardware used was an octa-core shared memory multicore computer comprised of two Intel® Xeon 5520, Quad-core, Nehalem 2.26 Ghz, with a 8 MB cache and a memory bandwidth of approximately 23 GB/s.

---

\(^2\)OpenMP (Open Multi-Processing) is an application programming interface that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran.
Table 1: Loss function (Eq. 22) values for sequential and parallel implementation of the KF for different problem sizes $N$.

<table>
<thead>
<tr>
<th>$N$</th>
<th>Sequential</th>
<th>Parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>$7.2987E-17$</td>
<td>$7.2583E-17$</td>
</tr>
<tr>
<td>100</td>
<td>$2.3841E-17$</td>
<td>$2.4012E-17$</td>
</tr>
<tr>
<td>200</td>
<td>$1.0021E-17$</td>
<td>$1.0531E-17$</td>
</tr>
<tr>
<td>300</td>
<td>$3.5315E-16$</td>
<td>$3.4714E-16$</td>
</tr>
<tr>
<td>400</td>
<td>$5.2381E-17$</td>
<td>$5.2190E-17$</td>
</tr>
</tbody>
</table>

Table 2: Single core execution time in milliseconds.

<table>
<thead>
<tr>
<th>$N$</th>
<th>BLAS</th>
<th>Alg. 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>7.18</td>
<td>0.61</td>
</tr>
<tr>
<td>100</td>
<td>10.12</td>
<td>2.09</td>
</tr>
<tr>
<td>200</td>
<td>29.89</td>
<td>8.08</td>
</tr>
<tr>
<td>300</td>
<td>92.65</td>
<td>18.01</td>
</tr>
<tr>
<td>400</td>
<td>167.99</td>
<td>31.64</td>
</tr>
</tbody>
</table>

Figure 2: Speed up curves for Alg. 2 for $N = 50$ to $N = 400$, using up to $M = 8$ processors. For reference linear speedup is marked by the dashed line.
Figure 3: Speedup curves for BLAS implementation for $N = 50$ to $N = 400$, using up to $M = 8$ processors. For reference linear speedup is marked by the dashed line.

9 Discussion

Even though each subroutine provided by BLAS is extremely efficiently implemented, the implementation of Alg. 2 executes faster on a single core, as can be seen from Tab. 2. This comes from the fact that an optimization over the sequence of operations can be made for Alg. 2, whereas the optimization is done over each single operation, one by one, when employing the BLAS implementation. When optimizing over sequences of operations, the possibility to make more efficient use of the memory hierarchies is a main factor in the faster execution.

Regarding scalability, the implementation of Alg. 2 performs far better than the BLAS implementation. As discussed previously, this is due to less communication and parallel overhead. The effects are especially distinct for smaller problem sizes where the overhead constitutes a large proportion of the execution time. For Alg. 2, almost linear speedup in the number of cores used is achieved for $N \geq 200$. For lower $N$, the gain of parallel implementation is less clear, and for $N = 50$ even a slowdown can be observed for $M = 8$, due to the disproportionally large overhead to computation ratio. However, for smaller problem sizes, not even 2 times speedup is reached for the BLAS implementation, and a slowdown can be observed for $N \leq 200$.

The implementations were evaluated on a hardware that runs at a very high CPU clock frequency (2.26 GHz). Embedded hardware, especially low power systems, typically run at much lower clock frequencies. With a lower clock
frequency, the scalability can be expected to be better for smaller problem sizes since the computation-to-overhead ratio goes down.

As mentioned before, Alg. 2 will most likely perform well on a distributed system. This is because of the low amount of communication, shared data, and the fact that only a restricted part of $P$ must be touched by each processor. This would definitely not be the case for the BLAS implementation that would require almost all data to be distributed over the whole network.

10 Conclusions

Parallel multicore implementation of the Kalman filter is studied. An implementation based on parallelization of each step using BLAS is compared to an implementation that exploits a banded structure of the system matrix. It is shown that for systems which can be realized with a banded system matrix, the KF can be almost completely parallelized with a very restricted amount of inter-processor communication. Application to a radio interference power estimation problem demonstrated a linear speedup in the number of cores used, for state numbers that are becoming relevant in smart-phone dominated traffic scenarios.

References


Paper III
Parallel Recursive Bayesian Estimation on Multicore Computational Platforms Using Orthogonal Basis Functions

Olov Rosén and Alexander Medvedev∗†

Abstract

A method to solve the recursive Bayesian estimation problem by making use of orthogonal series expansions of the involved probability density functions is presented. The coefficients of the expansion for the posterior density are then calculated recursively via prediction and update equations. The method has two main benefits: it provides high estimation accuracy at a relatively low computational cost and it is highly amenable to parallel implementation. An application to a bearings-only tracking problem shows that the proposed method performs with the same accuracy as the particle filter but at a 24 times lower computational cost. A parallel implementation on a shared-memory multicore machine demonstrates that linear speedup in the number of cores is achievable.

1 Introduction

Nonlinear non-Gaussian filtering problems arise in numerous signal processing and control applications such as communication, radar and sonar target tracking, and satellite navigation, to mention a few.

Consider a nonlinear discrete-time system

\[
\begin{align*}
    x_{k+1} &= f(x_k) + w_k, \\
    y_k &= g(x_k) + v_k,
\end{align*}
\]

with the state vector \( x_k \in \mathbb{R}^n \), the output vector \( y_k \in \mathbb{R}^p \) and the process and measurement noise \( w_k \in \mathbb{R}^n, v_k \in \mathbb{R}^p \), respectively. The probability density functions (PDFs) \( p(w_k), p(v_k) \) are assumed to be known but are allowed to

∗The authors are partially supported by the project "Computationally Demanding Real-Time Applications on Multicore Platforms" funded by Swedish Foundation for Strategic Research.

†O. Rosén and A. Medvedev are with the Department of Information Technology, Uppsala University, Uppsala SE-751 05, Sweden \{olov.rosen, alexander.medvedev\}@it.uu.se
have arbitrary form. A more general formulation would as well include an
deterministic input signal $u_k$ but it is omitted here for brevity. A corresponding
generalization of the formulae is straightforward.
The Recursive Bayesian Estimation (RBE) problem, see e.g. [1], [2], is the gen-
eral underlying problem for nonlinear state estimation in stochastic dynamical
systems. In RBE, the aim is to recursively obtain the PDF $p(x_k|y_{1:k})$ subject
to system model (1)-(2) and given the measurements

$$y_{1:k} = \{y_1, y_2, ..., y_k\}.$$  (3)

For linear systems with white Gaussian process and measurement noise, an
exact solution of RBE is given by the Kalman filter [3]. For more general
system structures, no analytical closed-form solution is generally available and
approximative approaches have to be resorted to. The most commonly used
methods fall into three main categories. Grid-based methods evaluate the in-
volved functions over a set of discrete grid points [4]. Numerical integration
methods, such as e.g. Gaussian quadrature, exploit numerical integration to
calculate the involved integrals [5]. Monte-Carlo methods, exemplified by the
extensively studied particle filter techniques [6], use Monte-Carlo simulation to
obtain a sample from the distribution of which the desired information can be
extracted. For a survey of these methods see e.g. [7],[8].

This paper investigates a solution to the RBE problem by expanding the PDFs
in question in series of orthogonal functions. Orthogonal bases have been for
a long time used in statistics to estimate PDFs with general distributions
[9],[10],[11],[12]. Using orthogonal expansions, one expects that it should be
possible to estimate the PDFs with a substantially lower number of variables
than e.g. in the particle filter or grid-based methods since the latter two meth-
ods extract their estimates from, loosely speaking, a histogram. Fig. 1 shows a
PDF approximated by a histogram based on a sample of size 100, compared to
approximations by means of the first ten Legendre and Fourier basis functions.
Even though the histogram uses 10 times more parameters, the estimation
accuracy of the series approximations is still superior. Since much fewer vari-
ables are required to approximate the PDF, a smaller computational load for
the same estimation accuracy can be expected when using orthogonal basis
expansions.

Known approaches to Bayesian estimation with orthogonal basis functions em-
ploy e.g. Fourier series [13] and wavelets [14]. Fourier basis functions definitely
present a suitable alternative. However, the compact support of the wavelets,
as further discussed in Sec. 6.3, poses problems in some applications by forc-
ing the number of terms required in the expansion to be relatively high. This
effectively undermines the low computational price of the method that is one
of its main benefits.

The development in the present paper is with respect to a general orthogonal
basis and is particularly focused on the favorable parallelization properties of
the method exemplified by an implementation on a shared memory multicore
Figure 1: Fourier, Legendre and histogram approximations of the PDF $p(x)$. The series expansions use the 10 first basis functions, and the histogram is based on 100 observations.

processor. No particular orthogonal basis is thus assumed and the choice of a suitable basis clearly depends on the PDFs in hand. However, the method yields an estimation algorithm with good parallelization properties, no matter what particular basis is used.

Since all high-performance and much of embedded hardware is nowadays based on parallel processing, parallelizability of algorithms is of outmost importance. The orthogonality of the basis functions helps to structure the computational workload into independent segments, a property that is sought for in parallel (or distributed) implementation of algorithms. This is in contrast to e.g. Gaussian sum filters [15], [7, 16], [5], where Gaussian density functions are used as basis functions. The Gaussian density functions do not enjoy the orthogonality property and hence yield estimation algorithms that cannot be parallelized efficiently because of the data dependencies in the calculations. On the contrary, efficient parallelizations of the Kalman filter and the particle filter on multicore platforms can be found in [17], [18].

To recapitulate the highlights of the proposed solution to the RBE problem: the use of orthogonal expansions yields a method that is relatively computationally cheap and can be very efficiently implemented on a parallel architecture.

The paper structure is as follows. In Sec. 2, the notation and assumptions made are summarized. The RBE problem is briefly reviewed in Sec. 3. In Sec. 4, the proposed method is presented followed in Sec. 6 by the practical issues that have to be dealt with in an implementation. An application to a bearings-only tracking problem as well as a speedup evaluation for a parallel implementation are given in Sec. 7. A discussion of the results is presented in Sec. 8, followed by the conclusions.
2 Notation and Assumptions

Let $\mathbb{R}$ and $\mathbb{N}$ be the set of real and natural numbers, respectively. Vectors are written in bold lower case and matrices are in bold upper case. The inner product over a domain $D$ is

$$< f(x), g(x) > = \int_D f(x)g(x) dx,$$  \hspace{1cm} (4)

while $\times$ denotes the Cartesian product. Concatenation of two vectors $n = (n_1, n_2, ..., n_d)$ and $m = (m_1, m_2, ..., m_d)$ is written as

$$nm = (n_1, n_2, ..., n_d, m_1, m_2, ..., m_d).$$

All series are assumed to be absolute converging.

2.1 Orthogonal functions

Let $T_i = [a_i, b_i] \subseteq \mathbb{R}$, $a_i < b_i$, $i = 1, 2, ..., d$ and

$$T^d = T_1 \times T_2 \times \cdots \times T_d.$$ 

Introduce an orthogonal basis $\Phi = \{\phi_n^{(i)}\}_{n=0}^{\infty}$ on $T_i$ with respect to the inner product in (4). A multivariate orthogonal basis on $T^d$ is given by $\{\phi_n(x)\}_{n \in \mathbb{N}^d}$, where $\phi_n(x)$ is interpreted as

$$\phi_n(x) = \prod_{i=1}^{d} \phi_{n_i}(x_i)$$

for the multi-index $n = (n_1, n_2, ..., n_d) \in \mathbb{N}^d$. From $L_2$ theory, any square integrable function $f(\cdot)$ on $T^d$ can be expressed as

$$f(x) = \sum_{n \in \mathbb{N}^d} c_n \phi_n(x),$$

where $c_n = < f(x), \phi_n(x) >$. The product of the $n$-th and $m$-th basis function will be assumed to have the expansion

$$\phi_n(x)\phi_m(x) = \sum_{k \in \mathbb{N}^d} g_{nmk} \phi_k(x).$$ \hspace{1cm} (5)

3 Recursive Bayesian estimation

Only a brief summary of RBE for the optimal filtering problem is provided in this section, see e.g. [4], [19],[16],[7] for a more extensive exposition. RBE
seeks to recursively obtain the PDF $p(x_k|y_{1:k})$ that specifies the conditional probability density for the state $x_k$ of the system (1), (2) at time step $k$, given the measurements $y_{1:k}$.

An iteration of the algorithm is comprised of two steps: prediction and update. Assume that $p(x_{k-1}|y_{k-1})$ is known. The predicted PDF $p(x_k|y_{k-1})$ is obtained by the Kolmogorov-Smirnov equation

$$p(x_k|y_{k-1}) = \int_{\mathbb{R}^d} p(x_k|x_{k-1})p(x_{k-1}|y_{k-1})dx_{k-1}. \quad (6)$$

When the measurement $y_k$ is acquired, the predicted PDF $p(x_k|y_{k-1})$ is updated from Bayes’ rule to give

$$p(x_k|y_k) = \frac{p(y_k|x_k)p(x_k|y_{k-1})}{p(y_k|y_{k-1})}. \quad (7)$$

In (6) and (7), $p(x_n|x_{n-1})$ and $p(y_n|x_n)$ are implicitly given by the system model. With $\delta(\cdot)$ denoting the Dirac delta, they can be obtained, via the generalized convolution formula as

$$p(x_k|x_{k-1}) = \int_{\mathbb{R}^d} \delta(x_k - f(x_{k-1}) - w_k)p_w(w_k)dw_k,$$

$$= p_w(x_k - f(x_{k-1}))$$

$$p(y_k|x_{k-1}) = \int_{\mathbb{R}^d} p(y_k - g(x_{k-1}) - v_k)p_v(v_k)dv_k,$$

$$= p_v(y_k - g(x_{k-1})).$$

In (7), $p(y_n|y_{n-1})$ is a normalizing constant and does not need to be explicitly evaluated in a practical implementation. Given an initial state $x_0$, an a priori PDF $p(x_0)$ and the measurements $y_{1:k}$, $p(x_k|y_k)$ can be obtained by applying (6) and (7) recursively.

4 Solving the RBE via series expansions

As there is no closed-form analytical solution for the general case of RBE numerical methods have to be employed to find an estimate of the state in (1), (2). The idea advocated here is to approximate the involved PDFs by orthogonal series expansions whose coefficients can be recursively determined from the prediction and update equations in (6)-(7).

Assume that $p(x_k|x_{k-1})$, $p(y_k|x_k)$ and $p(x_{k-1}|y_{k-1})$ are given by the expansions

$$p(x_k|x_{k-1}) = \sum_{n \in \mathbb{N}^d} \sum_{m \in \mathbb{N}^d} a_{nm}\phi_n(x_k)\phi_m(x_{k-1}), \quad (8)$$

$$p(y_k|x_k) = \sum_{n \in \mathbb{N}^d} \sum_{m \in \mathbb{N}^d} b_{nm}\phi_n(y_k)\phi_m(x_k), \quad (9)$$
\[ p(x_{k-1}|y_{k-1}) = \sum_{n \in \mathbb{N}^d} c_n \phi_n(x_{k-1}). \tag{10} \]

Inserting (8)-(10) into the prediction and update equations (6) and (7) yields the following relationships:

**Prediction step**

\[
p(x_k|y_{k-1}) = \int_{\mathbb{R}^d} p(x_k|x_{k-1})p(x_{k-1}|y_{k-1})dx_{k-1}
= \int_{\mathbb{R}^d} \left[ \sum_{n \in \mathbb{N}^d} \sum_{m \in \mathbb{N}^d} a_{nm} \phi_n(x_k) \phi_m(x_{k-1}) \times \sum_{k \in \mathbb{N}^d} c_k \phi_k(x_{k-1}) \right]dx_{k-1}
= \sum_{n \in \mathbb{N}^d} \sum_{m \in \mathbb{N}^d} \sum_{k \in \mathbb{N}^d} a_{nm} c_k \phi_n(x_k) \times \int_{\mathbb{R}^d} \phi_m(x_k) \phi_k(x_{k-1})dx_{k-1}
= \sum_{n \in \mathbb{N}^d} e_n \phi_n(x_k),
\]

with

\[ e_n = \sum_{m \in \mathbb{N}^d} \sum_{k \in \mathbb{N}^d} a_{n,m} c_k \cdot I_{mk}, \tag{11} \]

where

\[ I_{mk} = \int_{\mathbb{R}^d} \phi_m(x_k) \phi_k(x_{k-1})dx_{k-1}. \]

For real-valued basis functions, the integral above is equivalent to the inner product defined in (4) and hence

\[ I_{mk} = \begin{cases} 1 & \text{if } m = k, \\ 0 & \text{otherwise}. \end{cases} \tag{12} \]

For complex-valued basis functions, the integral is not equal to inner product (4). However, for the latter case, it holds that \( \phi_k(x) = \overline{\phi_{-k}(x)} \), and \( I_{mk} \) can be replaced by \( I_{m(-k)} \).

**Update step** When the measurement \( y_k \) becomes available, the PDF \( p(y_k|x_k) \) is conditionalized to yield

\[
p(y_k|x_k) = \sum_{n \in \mathbb{N}^d} \sum_{m \in \mathbb{N}^d} b_{nm} \phi_n(y_k) \phi_m(x_k)
= \sum_{m \in \mathbb{N}^d} f_m \phi_m(x_k),
\]
where

\[ f_m = \sum_{n \in \mathbb{N}^d} b_{nm} \phi_n(y_k). \]  

(13)

The multiplication in the update step is then carried out as:

\[
 p(x_k|y_k) = c p(y_k|x_k) p(x_k|y_{k-1}) \\
= c \sum_{n \in \mathbb{N}^d} f_n \phi_n(x_k) \sum_{m \in \mathbb{N}^d} e_m \phi_m(x_k) \\
= c \sum_{n \in \mathbb{N}^d} \sum_{m \in \mathbb{N}^d} f_n e_m \phi_n(x_k) \phi_m(x_k) \\
= c \sum_{n \in \mathbb{N}^d} \sum_{m \in \mathbb{N}^d} f_n e_m \sum_{k \in \mathbb{N}^d} g_{nmk} \phi_k(x_k) \\
= c \sum_{k \in \mathbb{N}^d} h_k \phi_k(x_k),
\]

where

\[
 h_k = \sum_{n \in \mathbb{N}^d} \sum_{m \in \mathbb{N}^d} e_m f_n g_{nmk}, \tag{14}
\]

and \( c \) is a normalizing constant given by

\[
 c = \left[ \sum_{n \in \mathbb{N}^d} h_n \int_{T^d} \phi_n(x_k) dx_k \right]^{-1}. \tag{15}
\]

The filtering can thus be performed by recursively calculating Eq. (11) and Eq. (14).

To recapitulate: Given the prior distribution

\[
p(x_{k-1}|y_{k-1}) = \sum_{n \in \mathbb{N}^d} a_n \phi_n(x_{k-1}),
\]

the steps required at one iteration of the algorithm to obtain the posterior distribution are:

- Perform the prediction step by evaluating (11).
- Calculate \( f_n, n \in \mathbb{N}^d \) from (13).
- Perform the update step by evaluating \( h_k, k \in \mathbb{N}^d \) from (14), and \( c \) from (15).
- The posterior distribution is now given from \( p(x_k|y_k) = c \sum_{n \in \mathbb{N}^d} h_n \phi_n(x_k). \)
5 Mean and Covariance

The mean and covariance for the PDF \( p(x_k|y_k) \) are typically of interest in estimation problems. The expected value in dimension \( i \) can be calculated by marginalizing the expansion for the \( i \)-th dimension and taking the expected value of the marginalized distribution, i.e.

\[
E[x_i|y_k] = \int_{T^d} x_i p(x_k|y_k) dx_k = \sum_{n \in \mathbb{N}^d} a_n \int_{T^d} x_i \phi_n(x_k) dx_k.
\]  

(16)

Let \( x_i \) denote the \( i \)-th element of \( x_k \). The covariance between \( x_i \) and \( x_j \) is given by

\[
cov(x_i, x_j) = E[(x_i - E[x_i])(x_j - E[x_j])] = E[x_i x_j] - E[x_i]E[x_j],
\]

where the second term is evaluated using (16), while the first term can be calculated as

\[
E[x_i x_j] = \int_{T^d} x_i x_j p(x_k|y_k) dx_k = \sum_{n \in \mathbb{N}^d} a_n \int_{T^d} x_i x_j \phi_n(x_k) dx_k.
\]  

(17)

6 Practical issues

The practical aspects that have to be dealt with in an implementation of RBE using orthogonal basis functions are discussed in this section.

6.1 Truncation

For an implementation, the infinite series must be truncated to some order \( N < \infty \), in each dimension. For simplicity of notation, it will be assumed that, in the multivariate case, the same approximation order is used for each dimension. The alternations in the formulae necessary for different approximation orders in different dimensions are straightforward.

In the update step (7), the order of the series expansion is doubled, in each dimension, due to the multiplication of series. Thus, to keep the order from growing exponentially, the series have to be truncated at each iteration. For simplicity, the truncation is made by keeping the first \( N \) terms. It should be noted that the truncation can result in an approximation \( \tilde{p}(x) \) that takes on
negative values, and is hence not a PDF. However the purpose of the approximation is to make inference about the state $x$, in this sense it is not worse to have $e(x) = \hat{p}(x) - p(x)$ negative than having $e(x)$ positive but merely $|e(x)|$ is of importance, as argued in [9].

### 6.2 Domain limitations

The discussion below concerns the one-dimensional case. For the multivariate case, the results hold for each dimension separately.

In general, the support of a PDF $p(x)$ is the whole real axis $\mathbb{R}$. However, for practical purposes, the PDF can be considered to have a compact support of $[a, b]$ if

$$ x \notin [a, b] \implies p(x) < \epsilon $$

for some user-defined value of $\epsilon$. The truncated support is denoted $\text{supp}_{\epsilon}(p(x)) = [a, b]$. Assume that the state and measurements are bounded $\forall k \geq 0$ as

$$ a_x \leq x_k \leq b_x, $$

$$ a_y \leq y_k \leq b_y, $$

and that, for a given $x_{k-1} \in [a_x, b_x]$, it holds that $\text{supp}_{\epsilon}(p(x_k|x_{k-1})) \subseteq [a_1, b_1]$. The approximation of $p(x_k|x_{k-1})$ must then be accurate over the domain

$$ \{(x_k, x_{k-1}) | a_x \leq x_k \leq b_x, a_x - a_1 \leq x_{k-1} \leq b_x + b_1\}. $$

Similarly, for a given $y_k \in [a_y, b_y]$ and $\text{supp}_{\epsilon}(p(y_k|x_k)) \subseteq [a_2, b_2]$, the approximation of $p(y_k|x_k)$ must be accurate over the domain

$$ \{(y_k, x_k) | a_y \leq y_k \leq b_y, a_y - a_2 \leq x_k \leq b_y + b_2\}. $$
6.3 Choice of basis functions

Since the computational burden will grow with the order of the series expansions $N$, it is highly desirable to use basis functions that allow for close approximations of the PDFs for as low $N$ as possible.

In general, $p(x_k|\bar{x}_{k-1})$ and $p(y_k|x_k)$ have unlimited support. In the estimation algorithm they are approximated offline over some finite domain. Using basis functions that decay rapidly near the domain borders, such as e.g. wavelets or Hermite functions, it can be challenging to give an accurate approximation with a low number of terms. In Fig. 2, the PDF $p(x_k|\bar{x}_{k-1})$ is plotted for the system $x_k = x_{k-1}^2 + w_{k-1}$, where $w_{k-1}$ is normally distributed. Approximations of the PDF by Fourier basis functions and Hermite basis functions, with $N = 15$ terms in each dimension, are also shown. As each Hermite function has the factor $e^{-x^2}$ and hence decays rapidly, it is difficult to capture the shape of $p(x_k|\bar{x}_{k-1})$ using few terms. [14] uses wavelets and indeed an approximation order of $N = 125$ is used in one dimension, whereas in the Fourier functions example in Sec. 7 only $N = 15$ is required to give a satisfactory tracking accuracy, for a similar problem.

6.4 Computational complexity and parallel implementation

The PDFs $p(x_k|x_{k-1})$ and $p(y_k|x_k)$ are assumed to be determined beforehand and offline.

At each iteration Eq. (11) and Eq. (14) have to be evaluated. The prediction step (11) requires $N^{2d}$ multiplications and $(N - 1)^{2d}$ additions. The computational cost of the update step in (14) is dependent on $g_{nmk}$. In many cases $g_{nmk}$ is zero, except for a few certain values of $n$ and $m$, which property will reduce the computational complexity of (14). However, if none of $g_{nmk}$ is zero, the update step requires $N^d$ multiplications and $N^d(N - 1)^d$ additions. The total flop count is then given by

$$c(N, d) = 3N^{2d} + 2N^d(N - 1)^d.$$ 

An implementation advantage with the series expansion approach is that all required computations consist only of summation of products. This algorithm therefore belongs to the class of so-called ”embarrassingly parallel algorithms”, and can straightforwardly be parallelized by assigning a sub-sum for evaluation to each processing unit.

More precisely, assume that $M$ processing units are available. If $N$ is a set of cardinality $N$, $N_m$ is a subset of $N$ of cardinality $N/M$ (assumed to be integer) and $N_m \cap N_n = \emptyset$, $\bigcup_{m=1}^{M} N_m = N$ then processing unit $m$ will calculate
the following quantities:

\[ e_n = \sum_{m \in N} a_{n,m} e_m, \quad n \in N_m, \]

\[ h_k^{(m)} = \frac{1}{a} \sum_{m \in N_m} \sum_{n \in N} e_m f_n g_{n,m,k}. \]

All the sub-sums are summed up by a single processing unit as a last step to yield

\[ h_k = \sum_{m=1}^{M} h_k^{(m)}. \]

7 Example

For illustration of the suggested estimation concept, a scalar but realistic numerical example demonstrating is given below. A bearings-only tracking problem with a severe non-linearity in the measurement equation is considered that is known to require nonlinear filtering to avoid divergence of the estimate [20].

An object is traveling along a straight line and noisy bearing measurements \( y_k \) of its position \( x_k \) are taken by a sensor, Fig. 3. The system model is given by

\[
x_{k+1} = x_k + w_k \\
y_k = \tan^{-1} x_k + v_k,
\]

where \( w_k \) is normally distributed with the mean \( \mu_w = 0 \) and the variance \( \sigma_w^2 = 0.01 \). The measurement noise \( v_k \) obeys the multi-modal PDF

\[
p_v(v) = \frac{P_1}{\sigma_{v1}\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{v-\mu_{v1}}{\sigma_{v1}})^2} + \frac{P_2}{\sigma_{v2}\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{v-\mu_{v2}}{\sigma_{v2}})^2}
\]

shown in Fig. 4, with \( P_1 = 0.2, P_2 = 0.8, \sigma_{v1} = 0.1, \sigma_{v2} = 0.05, \mu_{v1} = 0.1, \mu_{v2} = -0.05 \).
7.1 Solution using Fourier basis functions

This section presents a solution of the bearing-only tracking problem obtained by applying the Fourier basis functions \([21]\)

\[
\phi_n(x) = \frac{1}{\sqrt{2\pi}} e^{inx}, \quad |n| \leq \frac{N-1}{2}
\]

that are orthogonal over the interval \([-\pi, \pi]\). To modify the basis functions to be orthogonal over a general interval, a linear transformation of \(x\) can be applied.

The sought estimate is the expected value of the approximated PDF \(\hat{x}_k = E[x_k|y_k]\). From (16) and (17), the mean and covariance can be calculated as

\[
E[x_k] = \sum_{n=-N/2}^{N/2} a_n \varphi_n,
\]

\[
E[(x_k - E[x_k])^2] = \sum_{n=-N/2}^{N/2} \frac{1}{in\pi} a_n \varphi_n - E[x_k]^2,
\]

where \(\varphi_n\) is defined as

\[
\varphi_n = \int_{-\pi}^{\pi} x \phi_n(x) dx = \frac{1}{\sqrt{2\pi}} \int_{-\pi}^{\pi} xe^{inx} dx
\]

\[
= \begin{cases} 
0 & \text{if } n = 0, \\
(-1)^{n+1} \frac{\sqrt{2\pi} i}{n} & \text{otherwise}.
\end{cases}
\]

Since \(\phi_n(x)\phi_m(x) = \phi_{n+m}(x)\), it follows that

\[
g_{nmk} = \begin{cases} 
1 & \text{if } n + m = k, \\
0 & \text{otherwise}.
\end{cases}
\]
The above result reduces the computational complexity of Eq. (14). The approximation order of $N = 15$ is selected. Fig. 5 depicts the true state, the measurement and the estimated state. In Fig. 6, the sequence of estimated PDF:s $p(x_k|y_k)$ is shown for $k = 1, 2, ..., 10$.

### 7.2 Particle filter comparison

For reference, the suggested nonlinear estimation method was compared to a SIR particle filter (PF) [19] in the root mean square error

$$E_{RMSE} = \sqrt{\frac{1}{K} \sum_{k=0}^{K} (x_k - \hat{x}_k)^2}$$

and the floating point operation (flop) demand.

Using the approximation of 10 flops to generate a random number, and 50 flops to evaluate the exponential function, the particle filter implementation requires about $410N_p$ flops per iteration, where $N_p$ denotes the number of particles. Tab. 1 summarises the RMSE values and the flop count for different particle set sizes $N_p$. This can be compared to the RMSE=0.079 obtained by the series expansion filtering at a flop count of 1700 flops.

### 7.3 Execution time and speedup

The execution time and scalability for different problem sizes $N_T = N^d$ were studied. Tab. 2 shows the execution time for single core execution. Fig. 7
Figure 6: $p(x_k|y_k)$ plotted for $k = 1, ..., 10$

Table 1: RMSE and flop count for a SIR particle filter implementation.

<table>
<thead>
<tr>
<th>$N_p$</th>
<th>50</th>
<th>100</th>
<th>200</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E_{RMSE}$</td>
<td>0.094</td>
<td>0.078</td>
<td>0.078</td>
</tr>
<tr>
<td>flops</td>
<td>$2.05 \cdot 10^4$</td>
<td>$4.1 \cdot 10^4$</td>
<td>$8.1 \cdot 10^4$</td>
</tr>
</tbody>
</table>

shows the speedup $s(M)$, i.e.

$$s(M) = \frac{t_1}{t_M},$$

where $t_M$ is the execution time obtained employing $M$ processors. The program was written in C++ using OpenMP for parallelization and execution was performed on a shared memory multicore processor (Quad-core Intel® Xeon 5520, Nehalem 2.26 GHz, 8MB cache).

Table 2: Execution time for single core execution for different problem sizes.

<table>
<thead>
<tr>
<th>$N$</th>
<th>100</th>
<th>300</th>
<th>500</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution time</td>
<td>0.0021</td>
<td>0.0568</td>
<td>0.2625</td>
<td>2.1467</td>
</tr>
</tbody>
</table>
Figure 7: Speedup plots for different values of $N$. For reference linear speedup is marked by the dashed line.

8 Discussion

8.1 Estimation accuracy

In the comparison to a SIR PF, it can be seen from Tab. 1 that about $N_p = 100$ particles are required to yield the same tracking performance as the suggested method. The computational cost of the PF for that number of particles is $4.1 \cdot 10^4/1700 \approx 24$ times greater than the cost for the orthogonal series expansion approach.

8.2 Speedup

One of the main advantages of the algorithm is its amenability to parallel implementation, as can be seen from the speedup plot in Fig. 7. The evolution of the multi- and many-core architectures is advancing at an increasing rate, while the single-core ones have basically stopped evolving. Therefore, it is highly desirable that an algorithm meant for online execution has good parallelization properties. As mentioned, Gaussian sum filters cannot be efficiently implemented in parallel due to the strong dependencies between the basis functions.
8.3 Limitations

The main limitations of the algorithm are the assumptions formulated in (18), (19). If $p(x_k|x_{k-1})$ and $p(y_k|x_k)$ are to be determined offline, the intervals $[a_x, b_x]$ and $[a_y, b_y]$ must be small enough relative to $\text{ supp}_\epsilon(p(x_k|x_{k-1}))$ and $\text{ supp}_\epsilon(p(y_k|x_k))$. The PDFs $p(x_k|x_{k-1})$ and $p(y_k|x_k)$ will otherwise appear as "spikes" and will demand an unreasonable high approximation order to produce a good fit. If the expansions for the PDFs $p(x_k|x_{k-1})$ and $p(y_k|x_k)$ are updated online, restrictions (18), (19) can be dropped. Doing so will, however, require a large amount of online computation and, therefore, reduce the real-time feasibility of the method. Similar to most of the estimation techniques, the exponential growth of the computational complexity with the dimension is a limitation that confines the possible applications to relatively low-dimensional ones.

9 Conclusions

A novel method for solving the recursive Bayesian estimation problem is presented. The method expands the involved probability density functions in series of orthogonal basis functions. The coefficients of the expansion are recursively calculated at each iteration through a prediction and an update step. The approach has two main advantages. First, a high estimation accuracy can be obtained by a small computational effort. Secondly, the orthogonality properties allows the computations to be separated into independent segments, making the algorithm very amenable for parallel implementation. A possible drawback is mainly that the state should be confined to a restricted domain. It is also possible to not specify a domain restriction on the state, but the method will then require considerably greater computational power.

References


Non-parametric anomaly detection in trajectorial data

Olov Rosén, Alexander Medvedev, Daniel Jansson.

Abstract

A non-parametric and system model-free anomaly detection method is presented. It is applicable in situations where system information is available in the form of trajectory data and a normal behavior can be established beforehand from repeated trajectory realizations. The method is based on the estimation of statistical distributions from a set of observations describing a normal behavior. With the aid of these distributions and by applying outlier detection methods, it can be concluded whether or not a given trajectory realization is likely to be generated by a system similar to the one that has produced the data sets describing the normal behavior. The performance of the developed method is illustrated on vessel traffic data from the English channel and a set containing eye-tracking data for oculomotor function evaluation. Since the method is model-free, it is especially suitable for systems that are difficult to model appropriately and/or highly nonlinear.

Anomaly detection refers to detecting patterns in a given data set that do not conform to a well-defined normal behavior [1]. It serves as an automatic method to detect system abnormalities or faults that potentially require a counteraction.

Provided the system under consideration can be appropriately modeled, a plethora of model-based methods can be applied [2], [3], [4]. However, in many cases, the operation principles of the system are not sufficiently known to constitute the basis of a first-principles model. Further, for non-linear and non-Gaussian systems, the computational burden of estimating a black-box model from data can be prohibitively high.

In this paper, a model-free method for anomaly detection, applicable to systems observed via trajectorial data, is developed. The method is computationally light, applies to nonlinear as well as non-smooth systems and can be outlined as follows (for an illustration of the given concepts see Fig. 1). Assume that a system $S$ follows a normal (or reference), possibly vector-valued, trajectory $r(\tau)$, where $\tau \in \mathbb{R}$ is a function of time $t$ and/or the system state $x(t)$, i.e.

$$\tau = g(t, x(t)).$$

The right-hand side expression in (1) will be referred to as the context function. For a given $\tau$, $x(t)$ should thus ideally be $r(\tau)$, but being subject to
disturbances and system uncertainty, $x$ can be considered as random variable $X$ characterized by the distribution $G(\tau)$.

Assume that a set of $N$ observed repeated system trajectory realizations from $S$, $\Gamma = \{\gamma_1, \gamma_2, ..., \gamma_N\}$ is available where

$$\gamma_i = \left\{ x^{(i)}(t_0^{(i)}), x^{(i)}(t_1^{(i)}), ..., x^{(i)}(t_{n_i}^{(i)}) \right\}$$

(2)

denotes the $i$-th realization and $x^{(i)}(t_j^{(i)})$, $1 \leq j \leq n_i$, are the state values at $n_i$ different, possibly non-uniformly sampled, time instants, $t_0^{(i)} < t_1^{(i)} < ... < t_{n_i}^{(i)}$.

Now consider a realization $\gamma_0$. It is sought to determine whether or not $\gamma_0$ is produced by the same system (i.e. $S$) as the set $\Gamma$. From the data in $\Gamma$, the distribution $G(\tau)$ and the corresponding probability density function (PDF) $f_X(\tau, x)$ can be estimated. To statistically test whether or not $\gamma_0$ differs from $\Gamma$, an outlier test can be performed on $\gamma_0$ w.r.t. $G$. Depending on the degree of outlyingness, the hypothesis of $\gamma_0$ being produced by $S$ can be either rejected or accepted.

The developed method is applied to a set containing vessel traffic data from the English channel, with the aim to find deviating activities. Due to the large number of objects in the scene and the need of prompt response to detected abnormalities, computationally demanding algorithms are not practically feasible. Further, the method is applied to a set containing eye-tracking data, with the aim to detect individual differences in the oculomotor system. The oculomotor system is inherently difficult to model due to its complex nature and model-free methods are therefore highly relevant. Appealing results for both applications are obtained.

The main contribution of the paper is a non-parametric and model-free algorithm for anomaly detection in systems characterized by trajectorial data. A novel method for outlier detection from non-parametric density estimation is derived and embedded in the algorithm.

Solutions bearing similarities to the method proposed here can be found in e.g. [5], [4], [6], [7]. However, the present work provides a generalization of the existing approaches that brings about significant refinements and addresses some of the shortcomings of the existing algorithms. To mention some, the proposed method can handle general distributions, in contrast to only Gaussian ones, and realizations that are non-uniformly sampled and/or of unequal lengths.

The paper is composed as follows. First, in Section 2 through Section 4, the individual steps comprising the algorithm are presented. In Sec. 5, the steps are brought together and the complete algorithm is summarized. Applications to simulated data, vessel traffic data and eye tracking data are presented in Section 6. Section 7 gives a discussion of the results and is followed by conclusions.
1 Notation and assumptions

The smallest squared Euclidean distance between a $d$-dimensional trajectory $r(u), u \in \mathbb{R}$ and a point $p \in \mathbb{R}^d$ is denoted

$$D(p, r(u)) = \inf_u || r(u) - p ||^2.$$ 

Let $Z = \{ z_i \}_{i=0}^n$ be a discrete set of points. The function that linearly interpolates a curve between consecutive points in $Z$ is defined as

$$l(\omega, Z) := (z_{\lceil \omega \rceil} - z_{\lfloor \omega \rfloor})(\omega - \lfloor \omega \rfloor) + z_{\lfloor \omega \rfloor}, \ 0 \leq \omega \leq n,$$

where $\lceil \cdot \rceil$ and $\lfloor \cdot \rfloor$ are the ceiling and floor function, respectively. $\mathcal{N}(\mu, \Sigma)$ denotes the normal distribution with the mean $\mu \in \mathbb{R}^d$ and the covariance $\Sigma \in \mathbb{R}^{d \times d}$.

Pr($A$) is the probability of the random event $A$. For a random variable $X$, the probability density function (PDF) and cumulative density function (CDF) are denoted $f_X(x)$ and $F_X(x)$ respectively.

2 Outlier detection

In outlier detection, one is concerned with designing a decision rule to determine whether or not an observation $x_0$ is likely to come from the distribution $G$. A majority of the multivariate outlier detection literature focuses on the normal distribution, see e.g. [8]. Distance-based measures of outlyingness [9], require the PDF to be unimodal and symmetric. Therefore, a multivariate outlier test for an arbitrary distribution is proposed below.

Let $X$ be a $d$-dimensional random variable with distribution $G$. A statistical test with the null hypothesis

$$H_0 : x_0 \text{ is an observation from } G$$
can then be based on the following definition of the multivariate \( p \)-value, i.e. the probability of get an observation at least as "extreme" as \( x_0 \)

**Definition 1.** The multivariate \( p \)-value for \( x_0 \) is defined as as

\[
p(x_0) = \int_{R(x_0)} f_X(x) dx,
\]

where

\[
R(x_0) = \{ x | f_X(x) \leq f_X(x_0) \}.
\]

An outlier test can then be performed at a significance level \( \alpha \) as

accept \( H_0 \) if \( p(x_0) \geq \alpha \).

If it is only sought to reject or accept \( H_0 \), rather than determining a specific \( p \)-value, a threshold value \( \eta \) can be enforced, such that

\[
\Pr( f_X(X) \leq \eta ) = \alpha
\]

and \( H_0 \) is accepted at a significance level \( \alpha \) if \( f(x_0) \geq \eta \) and rejected otherwise.

If \( G \) is unknown, but a sample from it is available, an estimate \( \hat{f}_X \) of \( f_X \) can be calculated from the given sample, and substituted into (3) and (4) in place of \( f \), to yield an approximative outlier test. The estimates of \( p \) and \( R \) are denoted \( \hat{p} \) and \( \hat{R} \) respectively. If the estimate \( \hat{f}_X \) converges uniformly to \( f_X \) as \( N \to \infty \), \( \hat{R} \) is a consistent estimator of \( R \), and hence \( \hat{p} \) is a consistent estimator of \( p \). How to fit the PDF \( \hat{f} \) to sample data and evaluate (3) is discussed in the following subsection.

### 2.1 Distribution fitting and \( p \)-value calculation

If the sample is known to come from a parametric family of distributions, a parametric estimator is a natural choice. If no a priori information about the distribution from which the sample is generated is available, non-parametric methods have to be employed to find \( \hat{f}_X(x) \). Histogram estimators, kernel estimators, and orthogonal basis function estimators are typical examples of non-parametric techniques, see e.g. [10],[11].

Orthogonal basis function approximation is used in this paper because of the ability to capture the form of \( f_X(x) \) with a low number of terms. As shown in [12], the coefficients \( a_n \) in the orthogonal series expansion, can be unbiasedly estimated as

\[
\hat{a}_n = \frac{1}{N} \sum_{i=1}^{N} w(x_i) \phi_n(x_i),
\]

where \( \{ \phi_n(x) \}_{i=1}^{N} \) is set of orthonormal basis functions, orthogonal w.r.t. to the inner product

\[
< \phi_n(x), \phi_m(x) > = \int_{\Omega} w(x)^2 \phi_n(x) \phi_m(x) dx
\]
If the estimated PDF comes from a parametric family of distributions, the p-value can often be analytically determined. For example, if \( X \sim \mathcal{N}(\mu, \Sigma) \), it can be shown [13], that the p-value is obtained from

\[
p(x_0) = \chi^2_d((x_0 - \mu)^T \Sigma^{-1} (x_0 - \mu))
\]

where \( \chi^2_d(z) \) is the z-quantic function for the \( \chi^2 \) distribution with d-degrees of freedom. For a non-parametric estimator, it is generally not possible to evaluate (3) analytically but numerical methods have to be employed.

3 Temporal and spatial context function

The function \( g \) in (1) is often naturally defined from either a temporal or spatial context. A temporal context function \( g_T \) is suitable when the system follows a normal trajectory \( r \) that is an explicit function of time, i.e. \( r = r(t) \) and

\[
g(t, x(t)) = g_T(t).
\]

A spatial context function \( g_S \) is eligible when the system follows a normal trajectory \( r \) that is a function of a subvector, \( s(t) \) of the state \( x(t) \), i.e. \( r = r(s(t)) \) and

\[
g(t, x(t)) = g_S(s(t)) = \arg \inf_\tau ||r(\tau) - s(t)||^2.
\]  

(5)

Examples of such systems can be e.g. vehicles traveling along routes and roads, or industrial robots performing a repetitive movement. In case \( r \) is unknown, an estimate \( \hat{r} \) can be calculated from \( \Gamma \). Let \( s^{(i)}_j \) denote a subvector of \( \bar{x}^{(i)}_j \), and let

\[
S = \{s^{(i)}_j | 1 \leq i \leq M, 1 \leq j \leq n_i\}
\]

be the subset of all points in \( \Gamma \) that correspond to \( s_j \). By e.g the method described in [14], \( \hat{r}(\tau), 0 \leq \tau \leq 1, \tau \in \mathbb{R}, \) can be approximated as a fitted curve to the data in \( S \). The trajectory \( r \) in (5) is then be replaced by \( \hat{r} \).

4 Probability density function

For a specific value of \( \tau \), \( X \) will be a stochastic variable with the distribution \( G(\tau) \) and the corresponding PDF \( f_X(\tau, x) \). The latter is assumed to be a continuous function of \( \tau \). In practice, to estimate \( f_X(\tau, x) \) from the data in \( \Gamma \), \( f_X(\tau, x) \) is estimated for a discrete set of values \( \{\tau_k\}_{k=1}^n \). Let \( \bar{x}_i(t) = l(t, \gamma_i) \) denote the continuous linearly interpolated trajectory of the discrete data in \( \gamma_i \). Let \( T_i \) be the set of points in the trajectories \( \bar{x}_i, i = 1, 2, ..., N \), that have context function equal to \( \tau_i \), i.e.

\[
T_i = \{\bar{x}_i(t) | \tau_i = g(t, \bar{x}_i(t)), i = 1, 2, .., N\}.
\]
A PDF $\hat{f}_{X,i}(x)$ can then be fitted to the data in $T_i$ using some of the distribution fitting methods reviewed in Section 2.1. The piecewise linear continuous approximation $\hat{f}_{X}(\tau, x)$ of $f_{X}(\tau, x)$ is then defined as

$$\hat{f}_{X}(\tau, x) = l(\tau, \{\hat{f}_{X,i}(x)\}_{i=1}^{N}).$$

**Proposition 1.** For a given $\tau$, $\hat{f}_{X}(\tau, x)$ is a PDF.

**Proof.** A function is a PDF if is non negative and integrates to 1. For $\hat{f}_{X}(\tau, x)$ it holds that

$$\int_{\mathbb{R}^d} \hat{f}_{X}(\tau, x) dx = l(\tau, \{\hat{f}_{X,i}(x)\}_{i=0}^{N}) =$$

$$\left[ \int_{\mathbb{R}^d} \hat{f}_{X,[\tau]}(x) dx - \int_{\mathbb{R}^d} \hat{f}_{X,\lfloor \tau \rfloor}(x) dx \right](\tau - \lfloor \tau \rfloor) + \int_{\mathbb{R}^d} \hat{f}_{X,\lfloor \tau \rfloor}(x) dx = (1 - 1)(\tau - \lfloor \tau \rfloor) + 1 = 1$$

and that $\hat{f}_{X}(\tau, x) = \hat{f}_{X,[\tau]}(x)(\tau - \lfloor \tau \rfloor) + (1 - (\tau - \lfloor \tau \rfloor))\hat{f}_{X,\lfloor \tau \rfloor}(x) \geq 0$, since $(\tau - \lfloor \tau \rfloor) \geq 0, 1 - (\tau - \lfloor \tau \rfloor) \geq 0, \hat{f}_{X,[\tau]}(x) \geq 0$ and $\hat{f}_{X,\lfloor \tau \rfloor}(x) \geq 0$, and is hence a PDF.

## 5 Anomaly detection algorithm

Assume that a data set $\Gamma$ comprising system trajectories arising from tracking of $r$ by system $S$ is given. Perform the following steps to determine whether or not the realization $\gamma_0$ defined by (2) is likely to be generated by similar mechanisms as were the data in $\Gamma$:

- **Specify a context function** $\tau = g(t, x)$, chosen from the nature of the system, as discussed in Section 3.
- **Calculate the estimated PDF** $\hat{f}_{X}(\tau, x)$ from the data in $\Gamma$ following Section 4.
- **For each observation** $x^{(0)}(t_i), i = 0, 1, ..., n_0$:
  - **Determine the context** by $\tau_i^* = g(t_i, x^{(0)}(t_i))$
  - **Calculate the p-value** $p$, w.r.t. $\hat{f}_{X}(\tau_i^*, x)$.
- **Based on the obtained p-values**, $p_i, i = 0, 1, ..., n_0$ determine if $\gamma_0$ should be accepted or rejected.

## 6 Experimental results

The proposed method was implemented and evaluated for two different applications: anomaly detection in vessel traffic data, and anomaly detection in
eye-tracking data. Lilliefors test [15] was applied to the training data for both applications. For the vessel traffic data and the eye tracking data, the null hypothesis that the data in $\Gamma$ are normally distributed was rejected in $14.3\%$ and in $39.7\%$ of the cases, respectively. Therefore, Gaussian distribution was selected for the former application and an unspecified distribution was used for the latter.

### 6.1 Vessel traffic

Supervision of vessel traffic is of importance to detect potentially dangerous or suspicious activities such as accidents, engine failures, smuggling, drunkenness etc. Manual supervision is an expensive and tedious task, due to the rarely occurring anomalies and the typically large number of objects in a scene. Data recordings from the Automatic Identification System (AIS)\(^1\) of freight ships traveling in the English Channel were made for 72 hours. A total of 182 trajectories containing the longitude, latitude, speed and heading, were recorded. From these, $N = 100$ trajectories were used as the training data set $\Gamma$. It was assumed that $G \sim \mathcal{N}(\mu, \Sigma)$. A spatial context function $g_S(s)$, where $s$ contain the longitude and latitude, was employed. The method in [14] was employed to evaluate $\hat{r}$ from the data in $\Gamma$. The anomaly detection algorithm was then applied to the remaining 82 trajectories revealing aberrations that fall into three types (see Fig. 2):

- **Type 1:** Vessels going into a harbor.
- **Type 2:** Vessels going off the route direction.
- **Type 3:** Vessels that present a clearly abnormal behavior compared to other vessels at similar positions.

The anomaly scores for Type 2 anomalies were several orders of magnitude lower than the anomaly scores for the Type 3 anomalies.

### 6.2 Eye-tracking

There are different types of eye movement (the two most commonly mentioned being saccades and smooth pursuit) [16], all of which are governed by complex neuromuscular systems. Research has shown that various medical conditions, e.g. Parkinson’s Disease [17], affect the smooth pursuit system (SPS) negatively, motivating the search for accurate quantification methods, which could then be used as diagnostic or even staging tools. The oculomotor system is inherently difficult to model due to complex non-linear dynamics and it is therefore of interest to find a non-parametric approach to use as a supplement to model-based methods.

\(^{1}\text{Vessels over 300 gross tonnes transmit their longitude, latitude, speed, and course via the AIS system.}\)
Figure 2: Zoom-in of points classified as anomalous of type 1, 2 and 3 respectively. Trajectories are given by gray lines. \( \hat{r} \) is marked by the thick line. Points classified as anomalous are marked by plus signs.

6.2.1 Experiment

Three test subjects

- \( P_1 \): Healthy man, 26 years old
- \( P_2 \): Healthy man, 27 years old
- \( P_3 \): Healthy man, 54 years old

were put in front of a computer screen and asked to follow a gaze reference \( r(t_k) \) in the form of a moving dot on the screen designed to have suitable characteristics as given in [18]. Thus \( r(t_k) \) is the \( x-y \) position of the dot at time step \( k \). The \( x-y \) coordinates of the points on the computer screen which the test subject was looking at, when exposed to the reference, were recorded using a video-based eye tracker from Smart Eye AB, Sweden. The \( j \)-th recording for test subject \( P_i \) is denoted

\[
\gamma^{(j)}_{P_i} = \left\{ x^{(j)}_{P_i}(t_0), x^{(j)}_{P_i}(t_1), \ldots, x^{(j)}_{P_i}(t_n) \right\},
\]

where \( x^{(j)}_{P_i}(t_k) \) is the \( x-y \) position at which the test subject \( P_i \) is looking at time sample \( t_k \). \( P_1 \) tracked the reference 40 times, while \( P_2 \) and \( P_3 \) tracked the reference 5 times each.

Since the reference is a function of time, a temporal context function \( g_T(t) = t \) was employed. The PDFs \( \hat{f}(t_k, x) \) were estimated from the first 35 realizations from \( P_1 \), i.e. \( \{x^{(j)}_{P_i}(t_k)\}_{j=1}^{35} \) using the 25 first Hermite basis functions [19]. The \( p \)-values for the data in \( \{x^{(j)}_{P_1}(t_k)\}_{j=36}^{40}, \{x^{(j)}_{P_2}(t_k)\}_{j=1}^{5} \) and \( \{x^{(j)}_{P_3}(t_k)\}_{j=1}^{5} \) were
Figure 3: $p$-values for test subject $P_1$, $P_2$ and $P_3$ for each sampling instant.

evaluated w.r.t. $\hat{f}(t_k, x)$, $k = 1, 2, ..., 500$. The mean of the $p$-value for each test subject is displayed in Fig. 3.

7 Discussion

7.1 Vessel traffic data

The underlying causes to the behaviors classified as anomalous by the algorithm are not known to the authors. The anomalies of Type 3 definitely seem reasonable to raise warnings for since the behavior is clearly distinguished from the other vessels at similar positions. Whether the Type 2 anomalies should raise warnings or not is more difficult to judge. However the anomaly scores for these were low and could just be presented as a notification to the operator. The Type 1 anomalies should not raise warnings since these are not actual anomalies. Since a ship is supposed to broadcast over the AIS system what harbor she is heading for, these types of warnings can easily be suppressed by using that additional information. A closer visual examination of the data also reveals that there are no apparent anomalies that are not detected by the algorithm.

7.2 Eye tracking data

From Fig. 3, the method suggests that the oculomotor system of $P_3$ differs from the oculomotor system of $P_1$ and $P_2$. A possible explanation of this, is that $P_3$ have an age of 54 years while $P_1$ and $P_2$ are 26 years old and 27 years old, respectively. Aging is a factor known to impair the oculomotor system. $P_1$ has the highest $p$-values, which is to expect since the PDF $\hat{f}(t_k, x)$ was estimated from data generated by $P_1$, and the definition of “normal” is thus based on the oculomotor system of $P_1$. As this study only contains 3 test subjects it is not possible to make more insightful conclusions based on these results. Subsequent studies containing more test subjects will be performed, to
enable drawing statistically significant conclusions.

7.3 Limitations

The method requires that enough realizations are available to be able to accurately estimate the involved PDFs. This can be especially problematic for high-dimensional systems since estimation of PDFs of high dimension requires many observations to achieve accuracy. It has though been shown [20] that orthogonal series estimates exhibit a convergence rate that is independent of the dimension, which make them an appealing option for high-dimensional estimation.

8 Conclusions

A non-parametric and model-free anomaly detection method is presented. The method is applicable to systems following a given reference whose trajectory realizations are observed. The method is based on the estimation of statistical distributions characterizing the trajectory deviations from the reference. With the aid of these distributions and by utilizing outlier detection methods, it can be concluded whether or not a given system trajectory is likely to be generated by the same mechanisms as the training data set. The developed method is applied and performs well for two real data sets. Since the method is model-free, it is suitable for systems that are difficult to model appropriately and/or highly nonlinear.

References


Recent licentiate theses from the Department of Information Technology

2012-009 Andreas Sembrant: *Efficient Techniques for Detecting and Exploiting Runtime Phases*

2012-008 Palle Raabjerg: *Extending Psi-calculi and their Formal Proofs*

2012-007 Margarida Martins da Silva: *System Identification and Control for General Anesthesia based on Parsimonious Wiener Models*

2012-006 Martin Tillenius: *Leveraging Multicore Processors for Scientific Computing*

2012-005 Egi Hidayat: *On Identification of Endocrine Systems*

2012-004 Soma Tayamon: *Nonlinear System Identification with Applications to Selective Catalytic Reduction Systems*

2012-003 Magnus Gustafsson: *Towards an Adaptive Solver for High-Dimensional PDE Problems on Clusters of Multicore Processors*

2012-002 Fredrik Bjurefors: *Measurements in Opportunistic Networks*

2012-001 Gunnika Isaksson-Lutteman: *Future Train Traffic Control – Development and deployment of new principles and systems in train traffic control*

2011-006 Anette Löfström: *Intranet Use as a Leadership Strategy*

2011-005 Elena Sundkvist: *A High-Order Accurate, Collocated Boundary Element Method for Wave Propagation in Layered Media*

2011-004 Niclas Finne: *Towards Adaptive Sensor Networks*