2011 50th IEEE Conference on Decision and Control and
European Control Conference (CDC-ECC)
Orlando, FL, USA, December 12-15, 2011

# Efficient Parallel Implementation of a Kalman Filter for Single Output Systems on Multicore Computational Platforms

Olov Rosén, Alexander Medvedev

*Abstract*— **Parallelization and cache memory bandwidth demand of a Kalman filter for single output systems on multicore computers are investigated and exemplified by an adaptive filtering application. By breaking the data dependencies through a re-organization of calculations, an almost completely parallel algorithm is obtained. Analysis of the resulting algorithm brings about an estimate of the memory bandwidth necessary for a linear in the number of cores speedup. An evaluation of the parallel algorithm on two different shared-memory multicore architectures has been performed. It is found that linear speed-up in the number of used cores can indeed be achieved provided a sufficient memory bandwidth is offered by the hardware.**

## I. Introduction

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. Unfortunately, for $N$ states to be estimated, the computational complexity of the KF is $O(N^3)$ [1], 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 [2]. 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) [3], [4]. 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, the benefit of individually set time variation of states, see e.g. [5]. However since the RLS is a special case of the KF [6], the proposed parallelization also applies to the RLS.

The computer industry has now entered the multicore era with hardware computational capacity increased by adding more processors (cores) on one chip. All major manufactures of processor chips have moved to a multicore (many core) design and sequential processors will not be available already in near future [7]. 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) [8] [9].

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 slow-down 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 the processor 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.

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 [10], distributed memory machines [11] and systolic arrays [12] and thus can not be used for a multicore implementation. Other parallelization solutions suffer from the presence of sequentially executed sections that prevent significant speedup [13] [14]. Pipelined by design algorithms [15] 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 article known to the authors of this paper addresses parallelization and cache memory handling of the KF on multicore platforms. Notably, a similar analysis for Particle Filters is presented in [16].

In this article, 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 speed-up in the number of used cores. Analysis of the resulting algorithm brings about an estimate

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

of the memory bandwidth necessary for a realization of this potential on a multicore computer.

The article aims at bringing computer architecture insights into the design of signal processing algorithms and, therefore, a brief summary of the memory architecture for a multicore is given in Section IV, to facilitate the understanding of some points in the implementation. For a more extensive exposition of these concepts see e.g. [17]. In Section V, the plant and the corresponding KF equations are described and the parallel implementation is provided in Section VI. Results and discussion follow in Section VIII and Section IX, respectively. Finally, the conclusions are given in Section X.

## II. Notation

Assume that $\mathbf{A}$ is a matrix of size $m \times n$. The submatrix that lies in the rows of $\alpha \subseteq \{1,..,n\}$ and columns of $\beta \subseteq \{1,..,m\}$ is denoted $\mathbf{A}(\alpha, \beta)$. Further, $1:n \triangleq \{1, 2, ..., n\}$, and $\mathbf{A}(:, j)$ denotes the j-th column of $\mathbf{A}$

## III. 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 $p_{|}$ that must be executed sequentially and a portion $p_{||} = 1 - p_{|}$ that can be run in parallel. A parallelization also imposes an overhead $c(M)$. Let $t_M$ denote the execution time when executed using $M$ processors. It can be shown that the obtained speed up is given by

$$s(M) = \frac{t_1}{t_M} = \frac{1}{p_{|} + \frac{(1-p_{|})}{M} + c(M)}$$

From the above expression, it can be seen that a program can never achieve a greater speedup than $s(\infty) = \frac{1}{p_{|}}$ (assuming $c(M) = 0$). This is known as Amdahl's law [18]. It is therefore of great importance that there are no large sequentially executed portions in a parallel implementation. The term $c(M)$ is somewhat difficult to estimate. It accounts for such effects as amount of communication, memory bandwidth, number of synchronization points etc. and is highly dependent on the architecture on which the program is executed.

## IV. Computer memory architecture

### A. 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. 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 temporal and spatial locality in data usage for most applications. 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 can take hundreds of 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 already brought into the cache will be reused to complete as many calculations as possible in the algorithm before they are thrown out.

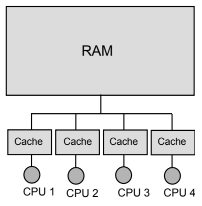

Fig. 1. Sketch of a shared-memory multicore memory architecture

Note that the picture 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 with up to three levels. However, to understand the ideas presented here, it suffice to think of the cache as a single memory with a much faster access time than the RAM memory.

### B. Shared memory multicore architecture

In a shared-memory multicore architecture, there are several CPUs that can access a shared RAM, however each CPU 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 actual organization of bus connections differs 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. 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.

## V. System model and the Kalman filter

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

$$\begin{aligned} \theta_{t+1} &= \theta_t + \epsilon_t \\ y_t &= \varphi_t^T \theta_t + e_t \end{aligned}$$

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

**Algorithm 1** Straightforward implementation of Eq. (1)-(3)

```
• $\mathbf{C}_t = \mathbf{P}_{t-1}\varphi_t$
• $b_t = \varphi_t^T \mathbf{C}_t$
• $d_t = r_t + b_t$
• $\mathbf{P}_t = \mathbf{P}_{t-1} + \mathbf{C}_t\mathbf{C}_t^T/d_t + \mathbf{Q}_t$
• $\hat{y}_t = \varphi_t^T \hat{\theta}_{t-1}$
• $\hat{\theta}_t = \hat{\theta}_{t-1} + \frac{\mathbf{C}_t}{d_t}[y_t - \hat{y}_t]$
```

$t-1$. The Kalman filter equations for estimation of $\theta_t$ (see e.g. [5]) are given by:

$$\hat{\theta}_t = \hat{\theta}_{t-1} + \mathbf{K}_t[y_t - \varphi_t^T\hat{\theta}_{t-1}] \quad (1)$$

$$\mathbf{K}_t = \frac{\mathbf{P}_{t-1}\varphi_t}{r_t + \varphi_t^T\mathbf{P}_{t-1}\varphi_t} \quad (2)$$

$$\mathbf{P}_t = \mathbf{P}_{t-1} - \frac{\mathbf{P}_{t-1}\varphi_t\varphi_t^T\mathbf{P}_{t-1}}{r_t + \varphi_t^T\mathbf{P}_{t-1}\varphi_t} + \mathbf{Q}_t \quad (3)$$

where $\hat{\theta}_t \in \mathbb{R}^N$ is the estimate of $\theta_t$, $\mathbf{K}_t \in \mathbb{R}^N$ is the Kalman gain, $\mathbf{P}_t \in \mathbb{R}^{N\times N}$ is the error covariance matrix, $r_t \in \mathbb{R}$ is the measurement noise variance $\mathrm{var}(e_t)$ and $\mathbf{Q}_t \in \mathbb{R}^{N\times N}$ is the covariance matrix of the process noise $\mathrm{cov}(\epsilon_t)$. *A priori* estimates of $\theta_0$ and $\mathbf{P}_0$ are taken as initial conditions, if available. Otherwise it is standard to use $\theta_0 = 0$ and $\mathbf{P}_0 = \rho I$ where $\rho$ is some "large" number.

A numerically sound alternative to (1) - (3) is the square root form known as the Square Root Information Filter (SRIF) [19]. The SRIF has a similar structure and data dependencies as (1) - (3) 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 (1) - (3) is treated further.

## VI. Implementation

In this section, computer implementation of the KF equations (1) - (3) 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 networked systems.

### A. Straightforward implementation

To minimize the computational redundancy in (1)-(3), the common terms $\mathbf{C}_t \triangleq \mathbf{P}_{t-1}\varphi_t$, $b_t \triangleq \varphi_t^T\mathbf{P}_{t-1}\varphi_t = \varphi_t^T\mathbf{C}_t$ and $d_t \triangleq r_t + \varphi_t^T\mathbf{P}_{t-1}\varphi_t = r_t + b_t$ are first calculated. This results in Alg. 1. The corresponding pseudo code is provided in Alg. 2.

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. $\mathbf{P}$ would then require $N^2(8\,B) = 32$ 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 $\mathbf{C}$ in Alg. 2, the elements of $\mathbf{P}_{t-1}$ will be brought into the cache as they are requested. Eventually, the elements of $\mathbf{P}_{t-1}$ that were first brought in will be substituted

**Algorithm 2** Pseudo code for implementation of Alg. 1

```
• for $i = 1:N$
    – for $j = 1:N$
        * $\mathbf{C}_t(i) = \mathbf{C}_t(i) + \mathbf{P}_{t-1}(i,j)\varphi_t(j)$
    – end
    – $b_t = b_t + \varphi_t(i)\mathbf{C}_t(i)$
    – $\hat{y}_t = \hat{y}_t + \varphi_t(i)\hat{\theta}_{t-1}(i)$
• end
• $d_t = r_t + b_t$
• for $i = 1:N$
    – for $j = 1:N$
        * $\mathbf{P}_t(i,j) = \mathbf{P}_{t-1}(i,j) + \mathbf{C}_t(i)\mathbf{C}_t(j)/d_t + \mathbf{Q}_t(i,j)$
    – end for
    – $\hat{\theta}_t(i) = \hat{\theta}_{t-1}(i) + \frac{\mathbf{C}_t(i)}{d_t}[y_t - \hat{y}_t]$
• end for
```

**Algorithm 3** Reorganized implementation of Alg. 1

```
• $d_t = r_t + b_t$
• $\hat{\theta}_t = \hat{\theta}_{t-1} + \frac{\mathbf{C}_t}{d_t}[y_t - \hat{y}_t]$
• $\mathbf{P}_t = \mathbf{P}_{t-1} + \mathbf{C}_t\mathbf{C}_t^T/d_t + \mathbf{Q}_t$
• $\mathbf{C}_{t+1} = \mathbf{P}_t\varphi_{t+1}$
• $\hat{y}_{t+1} = \varphi_{t+1}^T\hat{\theta}_t$
• $b_{t+1} = \varphi_{t+1}^T\mathbf{C}_{t+1}$
```

by the elements currently in use. When the program later arrives at the calculation of $\mathbf{P}_t$, the elements of $\mathbf{P}_{t-1}$ must be brought in once again. Since $\mathbf{P}$ is of considerable size, bringing it to the cache twice leads to a substantial increase in the execution time.

### B. 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 $\mathbf{P}$ is brought into the memory, it will be used to accomplish all calculations it is involved in. Therefore, squeezing the $\mathbf{P}$ matrix twice trough the memory at each iteration is no longer needed.

### C. Utilizing the symmetry of P

If $\mathbf{P}_0$ is symmetric, it can be seen from (3) that $\mathbf{P}$ will stay symmetric through the recursions. This should be taken advantage of, since approximately half of the calculations and memory storage can be spared. $\mathbf{C}_t(i)$ can be rewritten to be calculated from only upper triangular elements as

$$\mathbf{C}_t(i) = \sum_{j=i}^{N}\mathbf{P}_t(i,j)\varphi_t(j) + \sum_{j=1}^{i-1}\mathbf{P}_t(j,i)\varphi_t(j).$$

**Algorithm 4** Pseudocode of memory efficient implementation.

```
• $d_t = r_t + b_t$
• for $i = 1 : N$
    – $\hat{\theta}_t(i) = \hat{\theta}_{t-1}(i) + \frac{\mathbf{C}_t(i)}{d_t}[y_t - \hat{y}_t]$
    – for $j = 1 : N$
        ∗ $\mathbf{P}_{t+1}(i,j) = \mathbf{P}_t(i,j) + \mathbf{C}_t(i)\mathbf{C}_t(j)/d_t + \mathbf{Q}_t(i,j)$
        ∗ $\mathbf{C}_{t+1}(i) = \mathbf{C}_{t+1}(i) + \mathbf{P}_{t+1}(i,j)\varphi_{t+1}(j)$
    – end for
    – $\hat{y}_{t+1} = \hat{y}_{t+1} + \varphi^T_{t+1}(i)\hat{\theta}_t(i)$
    – $b_{t+1} = b_{t+1} + \varphi_{t+1}(i)\mathbf{C}_{t+1}(i)$
• end for
```

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

```
• for $j = i : N$
    – $\mathbf{P}_{t+1}(i,j) = \mathbf{P}_t(i,j) + \mathbf{C}_t(i)\mathbf{C}_t(j)/d_t + \mathbf{Q}_t(i,j)$
    – $\mathbf{C}_{t+1}(i) = \mathbf{C}_{t+1}(i) + \mathbf{P}_{t+1}(i,j)\varphi_{t+1}(j)$
    – $\mathbf{C}_{t+1}(j) = \mathbf{C}_{t+1}(j) + \mathbf{P}_{t+1}(i,j)\varphi_{t+1}(i)$
• end for
```

### D. Parallel implementation

Let $M$ be the number of CPUs used for the implementation. It can be observed by examining Alg. 4 that the only dependencies between $i$-loop iterations are in the adding up of $\hat{y}_{t+1}$, $b_{t+1}$ and $\mathbf{K}_{t+1}$. Such dependencies are easily broken by using a *reduction*, where each CPU calculates the local contribution to the sum that is later added up in a sequential section to give the global sum. 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 $\mathbf{P}$, 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 $\mathbf{P}$ to a rectangular matrix $\mathbf{P}'$ of size $N \times (N/2+1)$, where the mapping from an element in $\mathbf{P}$ to element $(i.j)$ in $\mathbf{P}'$ is given by

$$\mathbf{P}'(i,j) = \mathbf{P}(i,(i+j-1) \mod N), \qquad \begin{array}{l} 1 \le i \le N \\ 1 \le j \le (N/2+1) \end{array}$$

Notice that this matrix contains $N/2$ elements more than necessary. The upper triangular block of $\mathbf{P}$ contains $N(N+1)/2$ elements and $\mathbf{P}'$ 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 $\mathbf{P}'$ can be said to contain only upper diagonal elements since $\mathbf{P}(i,j) = \mathbf{P}(j,i)$.

$$\mathbf{P} = \begin{bmatrix} p_{11} & p_{12} & p_{13} & p_{14} & \cdot & \cdot \\ \cdot & p_{22} & p_{23} & p_{24} & p_{25} & \cdot \\ \cdot & \cdot & p_{33} & p_{34} & p_{35} & p_{36} \\ p_{41} & \cdot & \cdot & p_{44} & p_{45} & p_{46} \\ p_{51} & p_{52} & \cdot & \cdot & p_{55} & p_{56} \\ p_{61} & p_{62} & p_{63} & \cdot & \cdot & p_{66} \end{bmatrix} \to \mathbf{P}' = \begin{bmatrix} p_{11} & p_{12} & p_{13} & p_{14} \\ p_{22} & p_{23} & p_{24} & p_{25} \\ p_{33} & p_{34} & p_{35} & p_{36} \\ p_{44} & p_{45} & p_{46} & p_{41} \\ p_{55} & p_{56} & p_{51} & p_{52} \\ p_{66} & p_{61} & p_{62} & p_{63} \end{bmatrix}$$

The redundant elements of $\mathbf{P}'$ 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 $\mathbf{Q}$ to yield $\mathbf{Q}'$.

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

**Algorithm 5**

```
• Sequential
    – $\hat{y}_t = \sum_{m=1}^{M} y_t^{(m)}$
    – $b_t = \sum_{m=1}^{M} b_t^{(m)}$
    – $\mathbf{C}_t = \sum_{m=1}^{M} \mathbf{C}_t^{(m)}$
    – $d_t = r_t + b_t$
• CPU $m$ (in parallel)
    – for $i = i_{1m} : i_{2m}$
        ∗ $\hat{\theta}_t(i) = \hat{\theta}_{t-1}(i) + \mathbf{C}_t(i)/d_t[y_t - \hat{y}_t]$
        ∗ for $j = 1 : (N/2 + 1 - \lfloor \frac{2i}{N} \rfloor)$
            · $k = (i+j) \, mod \, N$
            · $\mathbf{P}'_{t+1}(i,j) = \mathbf{P}'_t(i,j) + \mathbf{C}_t^{(m)}(i)\mathbf{C}_t^{(m)}(k)/d_t + \mathbf{Q}'_t(i,j)$
            · $\mathbf{C}_{t+1}^{(m)}(i) = \mathbf{C}_{t+1}^{(m)}(i) + \mathbf{P}'_{t+1}(i,j)\varphi_{t+1}(k)$
            · $\mathbf{C}_{t+1}^{(m)}(k) = \mathbf{C}_{t+1}^{(m)}(k) + \mathbf{P}_{t+1}(i,j)\varphi_{t+1}(i)$
        ∗ end for
        ∗ $\hat{y}_{t+1}^{(m)} = \hat{y}_{t+1}^{(m)} + \varphi^T_{t+1}(i)\hat{\theta}_t(i)$
        ∗ $b_{t+1}^{(m)} = b_{t+1}^{(m)} + \varphi_{t+1}(i)\mathbf{C}_{t+1}^{(m)}(i)$
    – end for
```

## VII. Analysis of Algorithm 5

### A. 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.

### B. Communication and synchronization

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

The variables that are involved in a reduction, i.e. $\mathbf{C}$, $\hat{y}$ and $b$, which consists of $(N/2+1)+N/M+2$ elements, have to be communicated from the parallel 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 B per element), this means that for $N = 2000$, $(8\,B)(2000 + 3) \approx 16\,kB$ will need to be communicated, certainly not a large amount. The data to be communicated from the sequential to the parallel section are $\mathbf{C}$, $\hat{y}$, $b$ 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.

### C. *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 considerable size in the algorithm are $\mathbf{P}$ and $\mathbf{Q}$. Studying how these are transfered from the RAM to the CPU gives a good estimate of the required memory bandwidth. If the matrices $\mathbf{P}$ and $\mathbf{Q}$ have a size of $s(\mathbf{P})$ and $s(\mathbf{Q})$ bytes, transferring them from the RAM to the CPUs at each iteration requires a memory bandwidth of

$$B = \frac{[s(\mathbf{P}) + s(\mathbf{Q})] \cdot n_{\text{iter}}}{t_{\text{tot}}} \tag{4}$$

Even though $\mathbf{Q}_t$ 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. 4.

As for any other parallel algorithm, one could thus not expect this algorithm 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.

## VIII. Results

### A. *Hardware and software*

The algorithms were evaluated on two different computer architectures, Grad (Intel® 2.66GHz quad-core, E5430, 12 MB cache) and Kalkyl (quad-core Intel® Xeon 5520, Nehalem 2.26 GHz, 8 MB cache) [20]. 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 algorithms. Open MP [21] was used for parallelization which 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 $\mathbf{Q}$ 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 the memory bandwidth of Kalkyl and Grad were evaluated, to enable further analysis.

TABLE I

Execution times in sec. for 50 iterations of 2, Alg. 4 and Alg. 5, executed on a single core on Grad and Kalkyl.

| N | Grad | | | Kalkyl | | |
|---|---|---|---|---|---|---|
| | Alg. 2 | Alg. 4 | Alg. 5 | Alg. 2 | Alg. 4 | Alg. 5 |
| 500 | 0.12 | 0.063 | 0.021 | 0.12 | 0.051 | 0.028 |
| 1000 | 0.22 | 0.11 | 0.073 | 0.20 | 0.11 | 0.089 |
| 2000 | 1.06 | 0.60 | 0.33 | 0.99 | 0.56 | 0.34 |
| 4000 | 4.42 | 2.49 | 1.37 | 3.92 | 2.08 | 1.31 |
| 8000 | 17.55 | 9.60 | 5.51 | 16.52 | 8.45 | 5.54 |

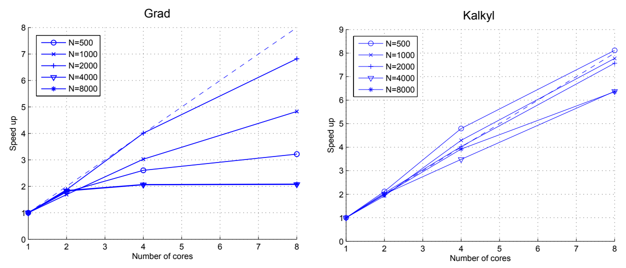

Fig. 2. Speedup for Alg. 5, executed on Grad (left) and Kalkyl (right). For reference linear speedup is marked by the dashed line.

### B. *Exection time and speedup*

Table I 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.

### C. *Memory bandwidth*

By means of the STREAM benchmark [22], which is specially designed to evaluate the memory bandwidth of a system, it was found that Grad has a memory bandwidth to the RAM of about 5.5 GB/s while Kalkyl has 23 GB/s. Tab. II show estimates of the required memory bandwidth $B_{\text{lin}}(N, M)$ to achieve linear speedup for problem size $N$ using $M$ processors. These values were obtained by applying (4) to the data in Tab.I, to calculate $B_{lin}(N, 1)$ with further extrapolation for $M \geq 1$, i.e $B_{lin}(N, M) = M \cdot B_{lin}(N, 1)$.

## IX. Discussion

It can be seen from Table I that the memory-efficient algorithm, Alg. 4, executes about twice as fast as the memory-

TABLE II

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

| M\N | 500 | 1000 | 2000 | 4000 | 8000 |
|---|---|---|---|---|---|
| Grad | | | | | |
| 1 | 2.4095 | 2.7255 | 2.4038 | 2.3229 | 2.3207 |
| 2 | 4.8190 | 5.4509 | 4.8075 | 4.6458 | 4.6414 |
| 4 | 9.6381 | 10.9019 | 9.6151 | 9.2915 | 9.2828 |
| 8 | 19.2762 | 21.8037 | 19.2302 | 18.5831 | 18.5657 |
| Kalkyl | | | | | |
| 1 | 1.7585 | 2.2470 | 2.3512 | 2.2636 | 2.3086 |
| 2 | 3.5169 | 4.4940 | 4.7023 | 4.5271 | 4.6173 |
| 4 | 7.0338 | 8.9881 | 9.4046 | 9.0542 | 9.2345 |
| 8 | 14.0677 | 17.9761 | 18.8093 | 18.1085 | 18.4690 |

inefficient algorithm, Alg. 2, on both systems (Grad and Kalkyl). Comparing execution times for Alg. 4 and Alg. 5 in Tab I, it can also be concluded that the execution time for the algorithm utilizing the symmetry of $\mathbf{P}$ runs, as expected, about twice as fast as the algorithm using the whole $\mathbf{P}$ 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 II where none of the elements exeedes the available bandwidth of 23 GB/s. Even super-linear 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.

*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 disproportionally large part of the execution time. Also in Tab. II, 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 VII-C assumes that $\mathbf{P}$ is transfered from the *RAM* to the CPU at each iteration. For $N \leq 2000$, the size of $\mathbf{P}$ satisfies $s(\mathbf{P}) \leq 16$ MB. Since there are 24 MB cache available running on 8 cores, the whole $\mathbf{P}$ 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 $N \geq 4000$, $s(\mathbf{P}) \geq 64$ 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. II 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.

## X. Conclusions

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. Analysis of the resulting algorithm yields an estimate of the memory bandwidth necessary for a linear in the number of cores speedup. 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. The bandwidth estimate gives a reasonable prediction of what memory system should be selected for the multicore platform and the estimation problem in hand in order to efficiently exploit the hardware.

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.

## References

[1] M. S. Grewal and A. P. Andrews, *Kalman Filtering : Theory and Practice Using MATLAB*, 2nd ed. Wiley-Interscience, Jan. 2001.

[2] E. Hansler, "The hands-free telephone problem: an annotated bibliography update," *Annals of Telecommunications*, vol. 49, pp. 360–367, 1994.

[3] T. Wigren, "Fast converging and low complexity adaptive filtering using an averaged Kalman filter," *Signal Processing, IEEE Transactions on*, vol. 46, no. 2, pp. 515 –518, Feb. 1998.

[4] M. Evestedt, A. Medvedev, and T. Wigren, "Windup properties of recursive parameter estimation algorithms in acoustic echo cancellation," *Control Engineering Practice*, vol. 16, no. 11, pp. 1372 – 1378, 2008.

[5] T. Söderström and P. Stoica, Eds., *System identification*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1988.

[6] L. Ljung and S. Gunnarsson, "Adaptation and tracking in system identification-a survey," *Automatica*, vol. 26, pp. 7–21, March 1990.

[7] "Multicore article series," http://web.mit.edu/newsoffice/2011/multicore-series-1-0223.html, Feb. 2011.

[8] "Intel core i7 extreme," "http://www.intel.com", Feb. 2011.

[9] "Amd phenom ii x6 black," http://www.amd.com/, Feb. 2011.

[10] M. Palis and D. Krecker, "Parallel Kalman filtering on the Connection Machine," in *Frontiers of Massively Parallel Computation, 1990. Proceedings., 3rd Symposium on the*, Oct. 1990, pp. 55 –58.

[11] P. M. Lyster, C. H. Q. Ding, K. Ekers, R. Ferraro, J. Guo, M. Harber, D. Lamich, J. W. Larson, R. Lucchesi, R. Rood, S. Schubert, W. Sawyer, M. Sienkiewicz, A. da Silva, J. Stobie, L. L. Takacs, R. Todling, and J. Zero, "Parallel computing at the nasa data assimilation office (dao)," in *Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM)*, ser. Supercomputing '97. New York, NY, USA: ACM, 1997, pp. 1–18.

[12] P. L. Shaffer, "Implementation of a parallel extended Kalman filter using a bit-serial silicon compiler," in *ACM '87: Proceedings of the 1987 Fall Joint Computer Conference on Exploring technology: today and tomorrow*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1987, pp. 327–334.

[13] D. Willner, C. B. Chang, and K. P. Dunn, "Kalman filter algorithms for a multi-sensor system," in *Decision and Control including the 15th Symposium on Adaptive Processes, 1976 IEEE Conference on*, vol. 15, Dec. 1976, pp. 570 –574.

[14] B. Tang, P. Cui, and Y. Chen, "A parallel processing Kalman filter for spacecraft vehicle parameters estimation," in *Communications and Information Technology, IEEE International Symposium on*, vol. 2, Oct. 2005, pp. 1476 – 1479.

[15] S. Howard, H.-L. Ko, and W. Alexander, "Parallel processing and stability analysis of the Kalman filter," in *Computers and Communications, 1996., Conference Proceedings of the 1996 IEEE Fifteenth Annual International Phoenix Conference on*, Mar. 1996, pp. 366 – 372.

[16] O. Rosen, A. Medvedev, and M. Ekman, "Speedup and tracking accuracy evaluation of parallel particle filter algorithms implemented on a multicore architecture," in *Control Applications (CCA), 2010 IEEE International Conference on*, Sep. 2010, pp. 440 –445.

[17] A. S. Tanenbaum and J. R. Goodman, *Structured Computer Organization*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1998.

[18] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. New York, NY, USA: ACM, 1967, pp. 483–485.

[19] G. J. Bierman, *Factorization Methods for Discrete Sequential Estimation*. New York, NY: Academic Press, 1977.

[20] "Uppmax," http://www.uppmax.uu.se, Aug. 2010.

[21] "Open mp," http://www.cs.virginia.edu/stream/, Aug. 2010.

[22] J. D. McClapin, "Stream benchmark," http://www.cs.virginia.edu/-stream/, Aug. 2010.