ELSEVIER

Available online at www.sciencedirect.com

SCIENCE DIRECT•

Computers and Mathematics with Applications 50 (2005) 881–893

An International Journal
computers & mathematics
with applications

www.elsevier.com/locate/camwa

Parallel Computations for Yau Filters

HON-WING CHENG AND STEPHEN S.-T. YAU
Control and Information Laboratory
Department of Mathematics, Statistics, and Computer Science
University of Illinois at Chicago
851 South Morgan Street (M/C 249)
Chicago, IL 60607-7045, U.S.A.
hwcheng@lucent.com yau@uic.edu

(Received October 2004; accepted January 2005)

Abstract—In the early 1990s, Yau developed a new class of nonlinear filters, called the *Yau Filters* which contain the Kalman-Bucy filters and the Benes filters as special cases. It has been shown that, from the Lie algebraic point of view, the Yau filters are the most general finite-dimensional filters. Yau and Hu proved that the DMZ equation for a Yau filter can be reduced to a Kolmogorov type partial differential equation and a system of linear differential equations. They noticed that the PDE is independent of the observed data and hence can be solved off-line. An efficient parallel algorithm for the system of ODEs would lead to fast solutions of Yau filters and hence their suitability for real-life applications. In this paper, we have proposed several parallel methods suitable for this system of ODEs. © 2005 Elsevier Ltd. All rights reserved.

Keywords—Yau filter, Ordinary differential equation, Initial value problem, Matrix exponential, Parallel algorithm.

1. INTRODUCTION

1.1. Notations

$\mathcal{R}$ denotes the set of all real numbers, and $\mathcal{R}^{n\times n}$ denotes the set of all real n-by-n matrices. We will use bold-faced italic lower-case letters (like $\boldsymbol{v}$) for vectors, and bold-faced italic upper-case letters (like $\boldsymbol{A}$) for matrices. $\boldsymbol{I}$ is the identity matrix of appropriate dimension. All logarithms will be of base 2 and will be denoted by lg.

Let $\boldsymbol{v}(t) = (v_i(t))$ be a vector-valued function of time t. The *derivative* of $\boldsymbol{v}$ is defined to be

$$\boldsymbol{v}'(t) := (v_i'(t)),$$

and the *integral* of $\boldsymbol{v}$ to be

$$\int \boldsymbol{v}(t)\,dt := \left(\int v_i(t)\,dt\right).$$

Research partially supported by Army Research Grant # DAAH 04-95-0530.
The parallel computing facilities for this work were provided by the Mathematics and Computer Science Division of the Argonne National Laboratory.

0898-1221/05/$ - see front matter © 2005 Elsevier Ltd. All rights reserved. Typeset by $\mathcal{A}_\mathcal{M}\mathcal{S}$-TEX
doi:10.1016/j.camwa.2005.01.026

For $\boldsymbol{X} \in \mathcal{R}^{n\times n}$, the *exponential* of $\boldsymbol{X}$ is defined by

$$e^{\boldsymbol{X}} := \sum_{k=0}^{\infty} \frac{\boldsymbol{X}^k}{k!}. \tag{1}$$

The following theorem is well known in differential equations [1].

THEOREM 1. *The unique solution to the following initial value problem:*

$$\begin{aligned} \boldsymbol{x}'(t) &= \boldsymbol{A}\boldsymbol{x}(t) + \boldsymbol{\beta}(t), \qquad t \geq t_0, \\ \boldsymbol{x}(t_0) &= \boldsymbol{x}_0, \end{aligned} \tag{2}$$

is given by

$$\boldsymbol{x}(t) = e^{(t-t_0)\boldsymbol{A}}\boldsymbol{x}_0 + \int_{t_0}^{t} e^{(t-s)\boldsymbol{A}}\boldsymbol{\beta}(s)\, ds. \tag{3}$$

1.2. The Yau Filter

In this paper, we will consider the following signal observation model:

$$\begin{aligned} dx(t) &= \boldsymbol{f}(\boldsymbol{x}(t)) + \boldsymbol{g}(\boldsymbol{x}(t))\, d\boldsymbol{v}(t), \qquad & \boldsymbol{x}(0) &= \boldsymbol{x}_0, \\ d\boldsymbol{y}(t) &= \boldsymbol{h}(\boldsymbol{x}(t)) + d\boldsymbol{w}(t), & \boldsymbol{y}(0) &= 0, \end{aligned} \tag{4}$$

where $\boldsymbol{x}$, $\boldsymbol{v}$, $\boldsymbol{y}$, and $\boldsymbol{w}$ are, respectively, $\mathcal{R}^n$, $\mathcal{R}^n$, $\mathcal{R}^m$, and $\mathcal{R}^m$ valued processes, and $\boldsymbol{v}$ and $\boldsymbol{w}$ have components which are independent standard Brownian processes. Furthermore, $\boldsymbol{f}(x)$ and $\boldsymbol{h}(x)$ are C^∞ vector-valued functions, and $\boldsymbol{g}(x)$ is orthogonal matrix-valued C^∞ function.

Model (4) arises from many practical problems. For example, it can be regarded as a dynamic system

$$\boldsymbol{x}'(t) = \boldsymbol{f}(\boldsymbol{x}(t))$$

subject to continuous random shock $\boldsymbol{v}(t)$ whose direction and intensity are modified by $\boldsymbol{g}(\boldsymbol{x}(t))$. We assume that the *state vector* (or *signal process*) $\boldsymbol{x}$ cannot be observed directly. However, a related quantity $\boldsymbol{y}$

$$\boldsymbol{y} = \boldsymbol{h}(\boldsymbol{x}(t))$$

can be observed, but it is corrupted by the noise $\boldsymbol{w}(t)$. The *filtering problem* is to find an optimal estimation of a certain function ϕ of $\boldsymbol{x}(t)$, based on all past observations $\boldsymbol{y}(s)$, $0 \leq s \leq t$.

Unfortunately, if the observations come in a high rate, then the computation may suffer these problems.

1. Long computing time.
2. Large memory requirement.

A *finite-dimensional recursive filter* (FDRF) is a filtering system which does not suffer the above drawbacks. The discussion of general FDRF is beyond the scope of this paper.

In 1990, Yau [2] first proposed the filtering system (4) with the following condition:

$$\frac{\partial f_j}{\partial x_i} - \frac{\partial f_i}{\partial x_j} = c_{ij}, \qquad \text{for } 1 \leq i, \quad j \leq n, \tag{5}$$

where c_{ij} are constants. The following two theorems were proved in [2].

THEOREM 2. *Condition (5) holds if and only if*

$$f_i(x) = \ell_i(x) + \frac{\partial F(x)}{\partial x_i}, \qquad 1 \leq i \leq n, \tag{6}$$

where $\ell_i = \sum_{j=1}^{n} d_{ij}x_j + d_i$, $1 \leq i \leq n$ *and* F *is a* C^∞ *function.*

If $F \equiv 0$ on $\mathcal{R}^n$, then (4) becomes the Kalman-Bucy filter; and if $\ell_i \equiv 0$ for all $i = 1, \ldots, n$, then (4) becomes the Benes filter.

THEOREM 3. *If the estimation algebra of the Yau filtering system is finite dimensional, then*

$$h_i(x) = \sum_{j=1}^{n} c_{ij}x_j + c_i, \qquad 1 \le i \le m, \tag{7}$$

where c_{ij} and c_i, $1 \le i, j \le n$, are constants.

Define

$$\eta(x) := \sum_{i=1}^{n} f_i^2(x) + \sum_{i=1}^{n} \frac{\partial f_i(x)}{\partial x_i} + \sum_{i=1}^{m} h_i^2(x). \tag{8}$$

We know that $\eta(x)$ is a polynomial of degree at most two for most interesting filtering systems [3,4]. Hence, we assume

$$\eta(x) = \sum_{i,j=1}^{n} \eta_{ij}x_i x_j + \sum_{i=1}^{n} \eta_i x_i + \eta_0, \tag{9}$$

where η_{ij}, η_i, $0 \le i, j \le n$, are constants.

Yau and Hu [5] showed that the DMZ equation for the finite-dimensional Yau filters satisfying (9) can be reduced to a Kolmogorov type partial differential equation and a system of ordinary differential equations as follows.

THEOREM 4. *Consider the Yau filtering system (4) satisfying conditions (6), (7), and (9). The solution $\xi(t,x)$ for the robust DMZ equation can be reduced to the solution $\tilde{\xi}(t,x)$ for the Kolmogorov type equation*

$$\frac{\partial \tilde{\xi}}{\partial t}(t,x) = \frac{1}{2}\Delta\tilde{\xi}(t,x) - \sum_{i=1}^{n} H_i(x)\frac{\partial \tilde{\xi}}{\partial x_i}(t,x) - P(x)\tilde{\xi}(t,x),$$

where

$$\tilde{\xi}(t,x) = e^{c(t)+G(x)+\sum_{i=1}^{n} a_i(t)x_i - F(x+b(t))}\xi(t, x+b(t))$$

and $a_i(t)$, $b_i(t)$ $(1 \le i \le n)$, and $c(t)$ satisfy the following system of ordinary differential equations:

$$a_i'(t) + \frac{1}{2}\sum_{j=1}^{n}(\eta_{ij}+\eta_{ji})b_j(t) + \sum_{j=1}^{n} d_{ji}b_j'(t) = 0, \tag{10}$$

$$b_i'(t) - a_i(t) - \sum_{j=1}^{n} d_{ij}b_j(t) + \sum_{j=1}^{m} c_{ji}y_j(t) = 0, \tag{11}$$

$$c'(t) + \frac{1}{2}\sum_{i=1}^{n}(b_i'(t))^2 - \sum_{i=1}^{n} a_i(t)b_i'(t) \tag{12}$$

$$-\sum_{i,j=1}^{n} \eta_{ij}b_i(t)b_j(t) - \sum_{i=1}^{n}\eta_i b_i(t) + \sum_{i=1}^{n} d_i b_i'(t) = 0, \tag{13}$$

with initial conditions $a_i(0) = b_i(0) = 0$ $(1 \le i \le n)$, and $c(0) = 0$, if we can choose $H(x)$, $G(x)$, and $P(x)$, such that

$$\sum_{i=1}^{n}\left(H_i^2(x) - \frac{\partial H_i(x)}{\partial x_i}\right) - \eta(x) + 2P(x) \equiv 0$$

and

$$H_i(x) - \frac{\partial G(x)}{\partial x_i} = \ell_i(x).$$

Several choices of $H(x)$, $G(x)$, and $P(x)$ were given in the same paper.

Note that the Kolmogorov equation (4) is independent of the observed data $y(t)$ and hence can be precomputed. The two systems of ODEs for $a(t)$ and $b(t)$ are special cases of the more general system (2). In fact, equations (10) and (11) fit in the linear system of ODEs (2) where $\beta(t)$ depends on the observation $y(t)$ of the filtering system (4). Function $c(t)$ can be computed by integration after $a(t)$ and $b(t)$ are solved.

In order to obtain real-time performance of the filtering systems, one needs to design efficient ODE solvers for (2). It is highly likely that parallel methods for systems of ODEs need to be introduced to provide acceptable real-time performance.

2. PARALLEL METHODS FOR ODES

In attempts to solve a general first-order initial value problem numerically, one has to generate sequence of state vectors $x(t_0), x(t_1), x(t_2), \ldots$ at the time mesh points $t_0 < t_1 < t_2 < \cdots$. Three types of parallelisms have been identified [6–9].

1. *Parallelism Across the Method*—Each $x(t_i)$ requires a number of function evaluations and one uses multiple processors to perform multiple function evaluations at the same time.
2. *Parallelism Across the System (Problem)*—Different processors are used to generate different components of $x(t_i)$ independently.
3. *Parallelism Across the Time (Steps)*—A sequence of state vectors $x(t_i), x(t_{i+1}), \ldots, x(t_{i+N-1})$ are generated concurrently by N processors.

For system (2), we have found that efficient algorithms can take elements from all three of these categories.

Our discussions will be from the pure algorithmic point of view, and hence, are independent of any specific parallel architecture. In particular, we make the following assumptions (cf. [10]).

1. The communication time among processors is negligible. (A shared memory architecture may satisfy this assumption.)
2. Each individual instruction takes the same time in sequential and parallel implementation.
3. If an instruction has to be executed k times, then the recurrent execution time on a single processor is k times larger than the concurrent execution time on k processors.

This way, the running time estimate is intrinsic to the parallel algorithm which represents the ideal performance attainable by any parallel machines.

2.1. Parallelism across the System

Many parallel algorithms on vectors use matrix-vector multiplication, vector addition, and scalar-vector multiplication as building blocks. A processor with a vector pipeline can perform these operations very efficiently. Without a vector pipeline, it is a simple exercise to employ multiple processors to compute different components of the state vectors [11,12], thus achieving parallelism across the system. To conceal this level of parallelism, we let $\alpha(n)$ and $\beta(n)$ be the running times for a matrix-vector multiplication and a vector addition (or scalar-vector multiplication), respectively, where a matrix is n by n and a vector has length n. We will use these as the units for measuring the running time of our parallel algorithms.

In this paper we assume the following.

1. The observation is sampled at equally spaced mesh points $t_0, t_1, \ldots, t_N$, i.e., $t_i - t_{i-1} = h$ for $i = 1, 2, \ldots, N$ where h is a positive constant.
2. To solve (2), we need to compute $x_i := x(t_i)$ for $i = 1, 2, \ldots, N$.
3. We consider $b(t_0), b(t_1), \ldots, b(t_N)$ as inputs to the system. The time to compute them is not included in our time analysis.
4. Our goal is to construct real-time solvers for (2). Any data that are not depending on the inputs can be precomputed and stored in the processors where they are needed. This will not be counted toward the running time of the algorithms.

2.2. Associative Fan-in Algorithm

Suppose we want to add $N = 2^m$ vectors of length n stored in N memory cells. The *associative fan-in algorithm* [13] exploits the associative property of the addition operation, and employs the following divide-and-conquer type strategy:

$$\sum_{i=1}^{N} x_i = \sum_{i=1}^{N/2} x_i + \sum_{i=1}^{N/2} x_{N/2+i},$$

and each of the terms on the right-hand side may likewise be computed recursively using the same strategy. As a result, the sum can be computed using $N/2$ processors in $((\lg N)\beta(n))$-time.

2.3. Quadrature Methods

The easiest way to introduce parallelism for system (2) is to exploit formula (3). In this paper, we assume the following about our system.

1. The observation is sampled at equally spaced mesh points $t_0, t_1, \dots, t_N$, i.e., $t_i - t_{i-1} = h$ for $i = 1, 2, \dots, N$ where h is a positive constant.
2. To solve system (2), we need to compute $x_i := x(t_i)$ for $i = 1, 2, \dots, N$.
3. We consider $b(t_0), b(t_1), \dots, b(t_N)$ as inputs to the system. The time to compute them is not included in our time analysis.
4. Since our goal is to construct on-time solvers for (2), any data that are not depending on the inputs can be precomputed and stored in the processors where they are needed. This will not be counted toward the running time of the algorithms.

In order that (3) can be applied effectively, we assume in this section that the matrix exponential e^{hA} and its powers can be accurately computed. (See [14,15].)

3. AN ALGORITHM FOR FUNCTION $c(t)$

The differential equation of $c(t)$ (12) can be written as

$$c'(t) = \Gamma(t), \qquad c(t_0) = c_0. \tag{14}$$

The solution can be easily obtained by an integration, i.e.,

$$c(t) = c_0 + \int_{t_0}^{t} \Gamma(s)\, ds. \tag{15}$$

When the sampling period is very small, we may use the *extended trapezoidal rule* for (4.2). It implies that $c_i := c(t_i)$ satisfies approximately

$$c_i = \sum_{k=0}^{i} \Gamma_k - \frac{1}{2}\Gamma_i, \qquad i \geq 1, \tag{16}$$

where

$$\begin{aligned} \Gamma_0 &:= c_0 + \frac{h}{2}\Gamma(t_0), \\ \Gamma_k &:= h\Gamma(t_k), \qquad 1 \leq k \leq i. \end{aligned} \tag{17}$$

Define

$$S_{j,k} := \sum_{\ell=j}^{k} \Gamma_\ell, \qquad 0 \leq j \leq k.$$

Equation (16) then becomes

$$c_i = S_{0,i} - \frac{1}{2}\Gamma_i. \tag{18}$$

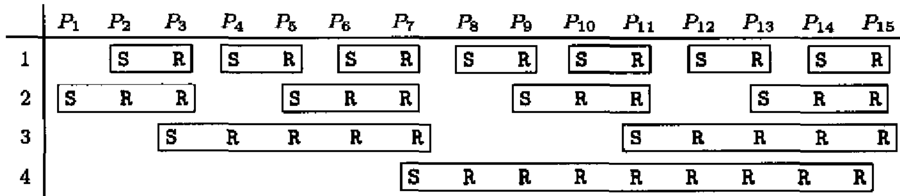

Figure 1.

Assume that $N = 2^m - 1$ processors: $P_1, P_2, \ldots, P_N$, are available. Let P_1 compute $S_{0,1} = \Gamma_0 + \Gamma_1$, and let P_k compute Γ_k according to (17), $1 < k \leq N$. Then the processors' values can be combined to yield $c_1, \ldots, c_N$ according to the *modified associative fan-in algorithm* depicted in Figure 1 ($N = 15$).

Each box above represents a group of communicating processors. 'S' denotes the sender and 'R' denotes a receiver. In each such group, the sender broadcasts its value to all receivers. A receiver adds the received value to its own value. For example, at iteration 1, P_2 sends its value (Γ_2) to P_3, and P_3 adds this value to its own (Γ_3). So after this transaction, P_3 has the value $S_{2,3}$. At iteration 2, P_1 sends its value $S_{0,1}$ to P_2 and P_3. After this transaction, P_2 has the value $S_{0,2}$ and P_3 has the value $S_{0,3}$. The values of the processors after each iteration are as shown in Table 1.

Table 1.

Processor	Initial Value	Iteration 0	1	2	3	4
P_1	$\Gamma_0 + \Gamma_1$	$S_{0,1}$				
P_2	Γ_2	$S_{2,2}$		$S_{0,2}$		
P_3	Γ_3	$S_{3,3}$	$S_{2,3}$	$S_{0,3}$		
P_4	Γ_4	$S_{4,4}$			$S_{0,4}$	
P_5	Γ_5	$S_{5,5}$	$S_{4,5}$		$S_{0,5}$	
P_6	Γ_6	$S_{6,6}$		$S_{4,6}$	$S_{0,6}$	
P_7	Γ_7	$S_{7,7}$	$S_{6,7}$	$S_{4,7}$	$S_{0,7}$	
P_8	Γ_8	$S_{8,8}$				$S_{0,8}$
P_9	Γ_9	$S_{9,9}$	$S_{8,9}$			$S_{0,9}$
P_{10}	Γ_{10}	$S_{10,10}$		$S_{8,10}$		$S_{0,10}$
P_{11}	Γ_{11}	$S_{11,11}$	$S_{10,11}$	$S_{8,11}$		$S_{0,11}$
P_{12}	Γ_{12}	$S_{12,12}$			$S_{8,12}$	$S_{0,12}$
P_{13}	Γ_{13}	$S_{13,13}$	$S_{12,13}$		$S_{8,13}$	$S_{0,13}$
P_{14}	Γ_{14}	$S_{14,14}$		$S_{12,14}$	$S_{8,14}$	$S_{0,14}$
P_{15}	Γ_{15}	$S_{15,15}$	$S_{14,15}$	$S_{12,15}$	$S_{8,15}$	$S_{0,15}$

After m iterations, processor i has the value $S_{0,i}$, $i = 0, 1, \ldots, N$. P_i then computes c_i according to (18).

ALGORITHM 1.

```
for (i ∈ {1,2,...,N}) do parallel {
    g_i ← f_i ← hΓ(t_i);
    if (i = 1) then
        g_1 ← c_0 + (h/2)Γ(t_0) + g_1;
}
for (k ← 1 to m) do
    for (j ∈ {2^{k-1} - 1, 2^{k-1} - 1 + 2^k, ..., 2^{k-1} - 1 + (2^{m-k-1} - 1)2^k}) do parallel
        // j = 0 is ignored in the first iteration.
```

```
        // Step size above is 2^k.
        for (ℓ ∈ {1,2,...,2^{k-1}}) do parallel
            g_{j+ℓ} ← g_{j+ℓ} + g_j;
for (i ∈ {1,2,...,N}) do parallel
    c_i ← g_i − ½f_i;
```

THEOREM 5. *Let γ be the time to evaluate $\Gamma(t)$ for any t. Using $N = 2^m - 1$ processors, Algorithm 1 can generate a sequence of N c_is in $\Theta(m\gamma)$-time.*

4. QUADRATURE METHODS FOR SYSTEMS OF EQUATIONS

The easiest way to introduce parallelism for (2) is to exploit (3). In order that (3) can be applied effectively, we assume in this section that the matrix exponential $e^{h\boldsymbol{A}}$ and its powers can be accurately computed. (See [14,15].)

4.1. Algorithm 2

With a constant sampling period h, the Newton-Cotes rules may be the most natural to apply. For example, consider the *Simpson's rule*

$$\int_{t_{i-2}}^{t_i} e^{(t_i-s)\boldsymbol{A}}\boldsymbol{b}(s)\,ds = \frac{h}{3}\left[e^{2h\boldsymbol{A}}\boldsymbol{b}(t_{i-2}) + 4e^{h\boldsymbol{A}}\boldsymbol{b}(t_{i-1}) + \boldsymbol{b}(t_i)\right] + O\left(h^4\right).$$

Let $\boldsymbol{E} := e^{h\boldsymbol{A}}$. It follows from (3) that $\boldsymbol{x}_i$ satisfies the following recurrence relation:

$$\boldsymbol{x}_1 = \boldsymbol{E}x_0 + \frac{h}{2}\left[\boldsymbol{E}b(t_0) + \boldsymbol{b}(t_1)\right] \qquad (\textit{trapezoidal rule}), \tag{19}$$

$$\boldsymbol{x}_i = \boldsymbol{E}^2\boldsymbol{x}_{i-2} + \frac{h}{3}\left[\boldsymbol{E}^2\boldsymbol{b}(t_{i-2}) + 4\boldsymbol{E}\boldsymbol{b}(t_{i-1}) + \boldsymbol{b}(t_i)\right], \qquad i \geq 2. \tag{20}$$

Note that $\boldsymbol{E}$ and $\boldsymbol{E}^2$ can be precomputed. The two sequences $(\boldsymbol{x}_0, \boldsymbol{x}_2, \boldsymbol{x}_4, \dots)$ and $(\boldsymbol{x}_1, \boldsymbol{x}_3, \boldsymbol{x}_5, \dots)$ can be generated in parallel, independently of each other. Hence using two processors, this algorithm can generate two state vectors in $(3\alpha(n)+4\beta(n))$-time. If this speed can keep pace with the input stream $\boldsymbol{b}(t_i)$, then (19) and (20) provide a reasonable real-time solver for system (2).

This algorithm can be improved by the following observations.

1. Parallelism across the method can be introduced by letting multiple processors perform the vector operations in parallel. For example, if eight processors are available, then six matrix-vector multiplications and two scalar-vector multiplications can be carried out concurrently and the results then added. The running time will be approximately $\alpha(n) + 2\beta(n)$ per two state vectors.
2. In general, a Newton-Cotes rule of order p can be written as follows:

$$\int_{t_i}^{t_{i+k}} \boldsymbol{f}(s)\,ds \approx h\sum_{j=0}^{k} w_j\boldsymbol{f}(t_{i+j}) + Ch^{p+1}\boldsymbol{f}^{(p)}(\xi),$$

where $w_0, \dots, w_k$, and C are constants, and ξ represents a value of s in the range of integration. Hence the recurrence for $\{\boldsymbol{x}_i\}$ has the following form:

$$\boldsymbol{x}_{i+k} = \boldsymbol{E}^k\boldsymbol{x}_i + h\sum_{j=0}^{k} w_j\boldsymbol{E}^{k-j}\boldsymbol{b}(t_{i+j}), \qquad i = 0, 1, \dots, k-1.$$

The subsequences $(\boldsymbol{x}_0, \boldsymbol{x}_k, \boldsymbol{x}_{2k}, \dots)$, $(\boldsymbol{x}_1, \boldsymbol{x}_{k+1}, \boldsymbol{x}_{2k+1}, \dots)$, $\dots$, $(\boldsymbol{x}_{k-1}, \boldsymbol{x}_{2k-1}, \boldsymbol{x}_{3k-1}, \dots)$ can be computed concurrently once the start-up state vectors $\boldsymbol{x}_0, \boldsymbol{x}_1, \dots, \boldsymbol{x}_{k-1}$ have been generated.

The previous discussions lead to the following algorithm.

ALGORITHM 2.

```
// Assume x_0,...,x_{k-1} are known, and E^k
// and hw_j E^{k-j}, j = 1,...,k, are precomputed
// and stored where they are needed.
for (ℓ ← 0 to N by k) do
    for (i ∈ {ℓ, ℓ+1,...,ℓ+k-1}) do parallel {
        for (j ∈ {0,1,...,k+1}) do parallel
            if (j = k+1) then
                d_{i+j} ← E^k x_i;
            else
                d_{i+j} ← hw_j E^{k-j} b(t_{i+j});
        x_{i+k} ← Σ_{j=0}^{k+1} d_{i+j}; // Use associative fan-in algorithm.
    }
```

THEOREM 6. *Ignoring the start-up time (time for generating $x_1,\ldots,x_{k-1}$), Algorithm 2 can generate k state vectors in $(\alpha(n) + \lceil\lg(k+2)\rceil\beta(n))$-time using $k(k+2)$ processors.*

4.2. Algorithm 3

Algorithm 2 achieves parallelism at the expense of more function evaluations per iteration. Its advantage is higher order of error estimate. However, the number of processors $k(k+2)$ required to gain the speed up is high. Furthermore, the start-up time is another problem to be reckoned with. So when the sampling period is very small, it may be better to use a lower order *Newton-Cotes rule* and introduce higher degree of parallelism across the time and across the method. For example, the classical *extended trapezoidal rule* for equation (3) reads

$$\int_{t_0}^{t_N} e^{(t_N - s)A} b(t)\, ds = \sum_{i=0}^{N} f_i + O\left(h^2\right), \tag{21}$$

where

$$\begin{aligned} f_0 &:= \frac{h}{2} E^N b(t_0), \\ f_i &:= h E^{N-i} b(t_i), \qquad 1 \le i < N, \\ f_N &:= \frac{h}{2} b(t_0). \end{aligned} \tag{22}$$

Using equation (21), equation (3) implies

$$x_{i+1} = E x_i + y_{i+1}, \qquad 0 \le i < N, \tag{23}$$

where

$$y_{i+1} := \frac{h}{2}\left(E b(t_i)) + b(t_{i+1})\right), \qquad 0 \le i < N. \tag{24}$$

If we let $y_0 := x_0$, then by iterating (24) repeatedly we obtain

$$u_i = \sum_{\ell=0}^{i} E^{i-\ell} y_\ell, \qquad i = 0, 1, \ldots, N. \tag{25}$$

Define

$$s_{i,j} := \sum_{\ell=i}^{j} E^{j-\ell} y, \qquad 0 \le i \le j \le N. \tag{26}$$

We observed that

$$\begin{aligned} y_j &= s_{j,j}, \\ x_j &= s_{0,j}, \end{aligned} \qquad 0 \le j \le N, \tag{27}$$

and

$$E^{j-i} s_{q,i} + s_{i+1,j} = s_{q,j}, \qquad 0 \le q \le i < j \le N. \tag{28}$$

Assume that $N = 2^m - 1$ and N processors, $P_1, P_2, \dots, P_N$, are available. Let P_1 compute $x_1 = Ey_0 + y_1$ and P_i compute y_i, $1 < i \le N$, initially. Using the same communication pattern as in Algorithm 1, the processors' values are combined as follows: in each communicating group, the sender broadcasts its value to all receivers. A receiver then multiplies the received value by an appropriate power of E and adds it to its own value. More specifically, in each iteration, suppose P_q sends its value $S_{i,q}$ to a receiver P_j which must have the value $S_{q+1,j}$. P_j will multiply the received value by E^{j-i} and adds it to its own value. Equation (28) implies that P_j will have the value $S_{q,j}$ after this transaction. The values of the processors after each iteration are depicted in Table 2.

Table 2.

	Initial	Iteration				
Proc.	Value	0	1	2	3	4
P_1	$Ey_0 + y_1$	$s_{0,1} = x_1$				
P_2	y_2	$s_{2,2}$		$s_{0,2} = x_2$		
P_3	y_3	$s_{3,3}$	$s_{2,3}$	$s_{0,3} = x_3$		
P_4	y_4	$s_{4,4}$			$s_{0,4} = x_4$	
P_5	y_5	$s_{5,5}$	$s_{4,5}$		$s_{0,5} = x_5$	
P_6	y_6	$s_{6,6}$		$s_{4,6}$	$s_{0,6} = x_6$	
P_7	y_7	$s_{7,7}$	$s_{6,7}$	$s_{4,7}$	$s_{0,7} = x_7$	
P_8	y_8	$s_{8,8}$				$s_{0,8} = x_8$
P_9	y_9	$s_{9,9}$	$s_{8,9}$			$s_{0,9} = x_9$
P_{10}	y_{10}	$s_{10,10}$		$s_{8,10}$		$s_{0,10} = x_{10}$
P_{11}	y_{11}	$s_{11,11}$	$s_{10,11}$	$s_{8,11}$		$s_{0,11} = x_{11}$
P_{12}	y_{12}	$s_{12,12}$			$s_{8,12}$	$s_{0,12} = x_{12}$
P_{13}	y_{13}	$s_{13,13}$	$s_{12,13}$		$s_{8,13}$	$s_{0,13} = x_{13}$
P_{14}	y_{14}	$s_{14,14}$		$s_{12,14}$	$s_{8,14}$	$s_{0,14} = x_{14}$
P_{15}	y_{15}	$s_{15,15}$	$s_{14,15}$	$s_{12,15}$	$s_{8,15}$	$s_{0,15} = x_{15}$

ALGORITHM 3.

// Assume the necessary powers of E have been
// precomputed and stored in each P_i.
for $(i \in \{1, \dots, N\})$ do parallel
    if $(i = 1)$
        $x_1 \leftarrow E(x_0 + \frac{h}{2} b(t_0)) + \frac{h}{2} b(t_1)$;
    else
        $x_i \leftarrow \frac{h}{2}\ (Eb(t_{i-1}) + b(t_i))$;
for $(k \leftarrow 1$ to $m)$ do
    for $(j \in \{2^{k-1} - 1, 2^{k-1} - 1 + 2^k, \dots, 2^{k-1} - 1 + (2^{m-k-1} - 1)2^k\})$ do parallel
        // $j = 0$ is ignored in the first iteration.
        // Step size above is 2^k.
        for $(\ell \in \{1, 2, \dots, 2^{k-1}\})$ do parallel
            $x_{j+\ell} \leftarrow x_{j+\ell} + E^\ell x_j$;

The initial step of P_i, $1 < i \le N$, takes $(\alpha(n) + 2\beta(n))$-time to complete. Note that the extra step taken by P_1 initially to compute x_1 does not increase the parallel time because it

does not participate in the first round of communication. After m iterations, processor i has the value x_{i+1}, $i = 0, 1, \dots, N-1$. These iterations will take a total of $m(\alpha(n) + \beta(n))$-time. Hence we have proved the following theorem.

THEOREM 7. *Using $N = 2^m - 1$ processors, Algorithm 3 can generate a sequence of N state vectors in $((m+1)\alpha(n) + (m+2)\beta(n))$-time.*

5. RUNGE-KUTTA METHOD

An explicit four-stage *Runge-Kutta* (*RK*) method with a fixed step size h for system (2) gives the following recurrence relation:

$$
\begin{aligned}
x_{i+2} &= x_i + \frac{1}{6}\left(k_0 + 2k_1 + 2k_2 + k_3\right), \qquad i \geq 0,\\
k_0 &:= 2h[Ax_i + b(t_i)],\\
k_1 &:= 2h\left[A\left(x_i + \frac{1}{2}k_0\right) + b\left(t_{i+1}\right)\right],\\
k_2 &:= 2h\left[A\left(x_i + \frac{1}{2}k_1\right) + b\left(t_{i+1}\right)\right],\\
k_3 &:= 2h\left[A\left(x_i + \frac{1}{2}k_2\right) + b\left(t_{i+2}\right)\right].
\end{aligned}
$$

It follows that

$$x_{i+2} = Mx_i + r_{i+2}, \tag{29}$$

where

$$M := I + \frac{h}{3}\left(6A + 5hA^2 + 3h^2A^3 + h^3A^4\right), \tag{30}$$

$$r_{i+2} := \frac{h}{3}\left\{\left(I + 2hA + 2h^2A^2 + h^3A^3\right)b(t_i) + \left(4I + 3hA + h^2A^2\right)b\left(t_{i+1}\right) + b\left(t_{i+2}\right)\right\}. \tag{31}$$

For simplicity of notation, assume $N = 2p = 2(2^m - 1)$. If we define $u_i^{(0)} := x_{2i}$ and $v_i^{(0)} := r_{2i}$, $i = 1, \dots, p$, then (29) becomes

$$
\begin{aligned}
u_0^{(0)} &= x_0,\\
u_{i+1}^{(0)} &= Mu_i^{(0)} + v_i^{(0)}, \qquad 0 \leq i < p.
\end{aligned} \tag{32}
$$

Note (32) is the same as (23), and hence, can be solved in the same way. Let us define

$$S_{i,j}^{(0)} := \sum_{\ell=i}^{j} M^{j-\ell} v_\ell, \qquad 0 \leq i \leq j \leq p. \tag{33}$$

Then $S_{i,j}^{(0)}$ satisfies

$$
\begin{aligned}
v_j^{(0)} &= S_{j,j}^{(0)},\\
u_j^{(0)} &= S_{0,j}^{(0)},
\end{aligned} \qquad 0 \leq j \leq p. \tag{34}
$$

Similarly, if we define $u_i^{(1)} := x_{2i+1}$ and $v_i^{(1)} := r_{2i+1}$, $i = 1, \dots, p$, then we have relationships analogous to equations (32)–(34).

Note that $(u_1^{(0)}, \dots, u_p^{(0)}) = (x_2, x_4, \dots, x_N)$, and $(u_1^{(1)}, \dots, u_p^{(1)}) = (x_3, x_5, \dots, x_{N+1})$. After generating x_1, say by a *predictor-corrector* procedure, the two subsequences can be computed independently using ideas similar to Algorithm 2. For example, let us assume $m = 3$, i.e., we use

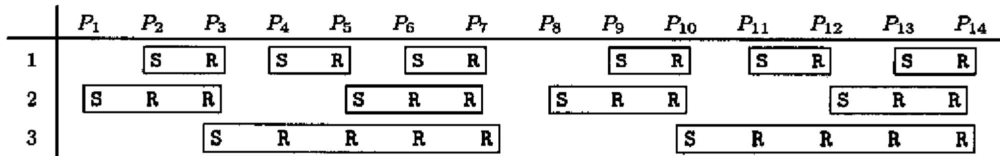

Figure 2.

Table 3.

Proc.	Initial Value	Iteration 0	Iteration 1	Iteration 2	Iteration 3	Final Value
P_1	$Mv_0^{(0)} + v_1^{(0)}$	$S_{0,1}^{(0)} = u_1^{(0)}$				x_2
P_2	$v_2^{(0)}$	$S_{2,2}^{(0)}$		$S_{0,2}^{(0)} = u_2^{(0)}$		x_4
P_3	$v_3^{(0)}$	$S_{3,3}^{(0)}$	$S_{2,3}^{(0)}$	$S_{0,3}^{(0)} = u_3^{(0)}$		x_6
P_4	$v_4^{(0)}$	$S_{4,4}^{(0)}$			$S_{0,4}^{(0)} = x_4^{(0)}$	x_8
P_5	$v_5^{(0)}$	$S_{5,5}^{(0)}$	$S_{4,5}^{(0)}$		$S_{0,5}^{(0)} = u_5^{(0)}$	x_{10}
P_6	$v_6^{(0)}$	$S_{6,6}^{(0)}$		$S_{4,6}^{(0)}$	$S_{0,6}^{(0)} = u_6^{(0)}$	x_{12}
P_7	$v_7^{(0)}$	$S_{7,7}^{(0)}$	$S_{6,7}^{(0)}$	$S_{4,7}^{(0)}$	$S_{0,7}^{(0)} = u_7^{(0)}$	x_{14}
P_8	$Mv_0^{(1)} + v_1^{(1)}$	$S_{0,1}^{(1)} = u_1^{(1)}$				x_3
P_9	$v_2^{(1)}$	$s_{2,2}^{(1)}$		$S_{0,2}^{(1)} = u_2^{(1)}$		x_5
P_{10}	$v_3^{(1)}$	$S_{3,3}^{(1)}$	$S_{2,3}^{(1)}$	$S_{0,3}^{(1)} = u_3^{(1)}$		x_7
P_{11}	$v_4^{(1)}$	$S_{4,4}^{(1)}$			$S_{0,4}^{(1)} = u_4^{(1)}$	x_9
P_{12}	$v_5^{(1)}$	$S_{5,5}^{(1)}$	$S_{4,5}^{(1)}$		$S_{0,5}^{(1)} = u_5^{(1)}$	x_{11}
P_{13}	$v_6^{(1)}$	$S_{6,6}^{(1)}$		$S_{4,6}^{(1)}$	$S_{0,6}^{(1)} = u_6^{(1)}$	x_{13}
P_{14}	$v_7^{(1)}$	$S_{7,7}^{(1)}$	$S_{6,7}^{(1)}$	$S_{4,7}^{(1)}$	$S_{0,7}^{(0)} = u_7^{(0)}$	x_{15}

$N = 2(2^3 - 1) = 14$ processors: $P_1, P_2, \dots, P_7$ will compute $x_2, x_4, \dots, x_{14}$, and $P_8, P_9, \dots, P_{14}$ will compute $x_3, x_5, \dots, x_{15}$. The communication patterns can be depicted as in Figure 2.

The values of the processors are combined using the same ideas as in Algorithm 3. The values after each iteration are depicted in Table 3.

The algorithm can be expressed as follows.

ALGORITHM 4.

// Assume the necessary powers of M have been
// precomputed and stored in each P_i.
Compute x_1 by a predictor-corrector method;
for $(\eta \in \{0,1\})$ do parallel {
    for $(i \in \{1,\dots,p\})$ do parallel {
        Compute $v_i^{(\eta)}$ according to (31);
        if $(i = 1)$
            $S_1^{(\eta)} \leftarrow Mx_0 + v_1^{(\eta)}$;
        else
            $S_i^{(\eta)} \leftarrow v_i^{(\eta)}$;
    }
    for $(k \leftarrow 1$ to $m)$ do
        for $(j \in \{2^{k-1} - 1, 2^{k-1} - 1 + 2^k, \dots, 2^{k-1} - 1 + (2^{m-k-1} - 1)2^k\})$ do parallel
            // $j = 0$ is ignored in the first iteration.

```
        // Step size above is 2^k.
        for (ℓ ∈ {1,2,...,2^(k-1)}) do parallel
            S_{j+ℓ}^{(η)} ← S_{j+ℓ}^{(η)} + M^ℓ S_j^{(η)};
}
```

Initially, it takes each processor $(2\alpha(n) + 3\beta(n))$-time to compute $v_i^{(0)}$ or $v_i^{(1)}$, $1 < i \leq p$. The extra steps required to compute $v_1^{(0)}$ and $v_1^{(0)}$ do not increase the parallel time since the corresponding processors do not participate in the first round of communication. Each iteration in the modified associative fan-in algorithm takes $(\alpha(n) + \beta(n))$-time to complete. The even and odd subsequences can be handled independently. Hence we have the following theorem.

THEOREM 8. *Let $N = 2(2^m - 1)$. After computing x_1, Algorithm 4 can generate N state vectors in $((m+2)\alpha(n) + (m+3)\beta(n))$-time using N processors.*

6. CONCLUSION

Yau and Yau introduced in [4] and then Yau and Hu in [16] a direct method for Kalman-Bucy filter with arbitrary initial condition. The same technique was later found to be applicable to the Yau filtering system. This novel approach is compared very favorably to other methods. It calls for the solution of a Kolmogorov type equation and a system of $2n + 1$ ODEs, where n is the length of the state vector. Since the Kolmogorov equation is independent of observation, it can be solved off-time. To solve the system of ODEs, we found that efficient algorithms can draw elements from all three categories of parallelism: *across the system, across the method*, and *across the time.* In this paper, we have proposed one parallel algorithm for solving $c(t)$ and three efficient parallel algorithms for solving the systems of $a(t)$ and $b(t)$. Their time analysis was also given.

The algorithm for $c(t)$ is in fact a parallel integration over a large number of equally spaced meshed points where many computations need to be performed. As for the system of ODEs, Algorithms 2 and 3 exploit an explicit formula while Algorithm 4 is a parallelized explicit Runge-Kutta method. If the matrix exponential e^{hA} and its powers can be accurately computed, Algorithms 2 or 3 should be chosen since they give more accurate solutions. Algorithm 3 is more efficient than Algorithm 4 and it should be the choice if h is very small and real-time performance is necessary. However, if h is relatively large and higher order of error estimate is required, then Algorithm 2 with order $p > 2$ should be needed. The performance of Algorithm 4 is similar to that of Algorithm 3, but it does not require the computation of matrix exponentials. Preliminary experiments showed that Algorithms 3 and 4 run at about the same time, and they are more efficient than Algorithm 2.

Our analysis has been purely algorithmic. It is independent of any parallel architecture. We have implemented our algorithms to run on the SP2 at the Argonne National Laboratory using *message passing interfaces* (*MPI*) for portability. The SP2 consists of 128 nodes, each of which is essentially an RS/8000 model with a 62.5 MHz clock, and two compiler servers. The nodes are connected via a high speed switch. Our current MPI implementation certainly does not take full advantage of these algorithms since message passing is relatively expensive. We expect better speed up on shared-memory machines. On the other hand, we treated the nonhomogeneous term $b(t)$ as input to the system. But for the system of ODEs arising from the filtering problems, the time to compute the nonhomogeneous term is $O(mn)$. Hence when m and n are large, the speed up of our algorithms will be much better. To simulate this behavior, a relatively expensive function $b(t)$ was used. Using 15 processors, we have observed a speed up of six. Further experiments will be carried out to study their performances on various parallel architectures.

REFERENCES

1. E.A. Coddington and N. Levinson, *Theory of Ordinary Differential Equations*, McGraw-Hill, (1955).

2. S.S.-T. Yau, Finite dimensional filters with nonlinear drift I: A class of filters including both Kalman-Bucy filters and Benes filters, *J. Math. Sys. Est. Cont.* (2), 181–203, (1994).
3. L.F. Tam, W.S. Wong and S.S.-T. Yau, On a necessary and sufficient condition for finite dimensionality of estimation algebras, *SIAM J. Control and Optimization* (1), 173–185, (1990).
4. S.S.-T. Yau and S.T. Yau, New direct method for Kalman-Bucy filtering system with arbitrary initial condition, In *Proceedings of 33rd CDC at Lake Buena Vista, Florida*, pp. 1221–1225, (Dec. 1994).
5. S.S.-T. Yau and G.Q. Hu, New direct method for Yau filtering system with arbitrary initial condition, In *Proceedings of the 35th Conference on Decision and Control*, Kobe, Japan, pp. 2539–2544, (1996).
6. M.N. El-Tarazi, Iterative methods for systems of first order differential equations, *IMA J. Numer. Anal.*, 29–39.
7. C.W. Gear, The potential for parallelism in ordinary differential equations, Report No. UIUCDCS-R-86-1246, Dept. of Computer Science, University of Illinois, (1986).
8. C.W. Gear, Parallel methods for ordinary differential equations, *Calcolo* **XXV** (1), (1988).
9. I. Lie, Some aspects of parallel Runge-Kutta methods, Mathematics and Computation 3/87, Dept. of Numerical Mathematics, Univ. of Trondheim, (1987).
10. A. Bellen, Parallelism across the steps for difference and differential equations, In *Lecture Notes in Mathematics*, pp. 22–35, (1989).
11. G.H. Golub and C. Van Loan, *Matrix Computations*, Second edition, The Johns Hopkins University Press, Baltimore, (1989).
12. S. Lakshmivarahan and S.K. Dhall, *Analysis and Design of Parallel Algorithms*, McGraw-Hill, (1990).
13. D. Heller, A survey of parallel algorithms in numerical algebra, *SIAM Review* **20**, 740–777, (1978).
14. C. Van Loan, The sensitivity of the matrix exponential, *SIAM J. Numer. Anal.* 14 (6), 971–981, (1977).
15. C. Moler and C. Van Loan, Nineteen dubious ways to compute the exponential of a matrix, *SIAM Review* **20**, 801–836, (1978).
16. S.S.-T. Yau and G.Q. Hu, Direct method without Riccati equation for Kalman-Bucy filtering system with arbitrary initial condition, In *IFAC 13th World Congress*, San Francisco, CA, June 30–July 5, (1996).
17. H.W. Cheng and S.S.-T. Yau, Parallel ODE-solvers for Kalman-Bucy filter with arbitrary initial condition, In *Proceedings of the 35th Conference on Decision and Control*, Kobe, Japan, pp. 4138–4145, (1996).
18. R. Vermiglio, A. Bellen and M. Zennaro, Parallel ODE-solvers with stepsize control, *J. Comput. Appl. Math.* (2), 277–293, (1990).
19. A. Bellen and M. Zennaro, Parallel algorithms for initial-value problems for difference and differential equations, *J. Comput. Appl. Math.* (3), 341–350, (1989).
20. W.H. Press *et al.*, *Numerical Recipes*, Cambridge University Press, (1986).
21. S.P. Nørsett and H.H. Simonsen, Aspects of parallel Runge-Kutta methods, In *Lecture Notes in Mathematics*, pp. 103–117, (1989).