Parallel Kalman Filtering On The Connection Machine

MS-CIS-90-81
LINC LAB 186

Michael A. Palis
University of Pennsylvania

Donald K. Krecker
General Electric Company

Department of Computer and Information Science
School of Engineering and Applied Science
University of Pennsylvania
Philadelphia, PA 19104-6389

November 1990

PARALLEL KALMAN FILTERING ON THE CONNECTION MACHINE

Michael A. Palis
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104
(215) 898-0376
palis@cis.upenn.edu

and

Donald K. Krecker
Advanced Technology Laboratories
General Electric Company
Moorestown, NJ 08057
(609) 866-6536
dkrecker@atl.ge.com

KEYWORDS: Kalman filtering, parallel algorithms, SIMD array architectures, Connection Machine

ABSTRACT: A parallel algorithm for square root Kalman filtering is developed and implemented on the Connection Machine (CM). The algorithm makes efficient use of *parallel prefix* or *scan* operations which are primitive instructions in the CM. Performance measurements show that the CM filter runs in time linear in the state vector size. This represents a great improvement over serial implementations which run in cubic time. A specific multiple target tracking application is also considered, in which several targets (e.g., satellites, aircrafts and missiles) are to be tracked simultaneously, each requiring one or more filters. A parallel algorithm is developed which, for fixed size filters, runs in constant time, independent of the number of filters simultaneously processed.

1. INTRODUCTION

Since the appearance of Kalman's original paper [12], the Kalman filter has become a fundamental tool for solving state estimation problems. Kalman filtering is a general technique for recursively estimating the state variables of a dynamic system from noisy observations or measurements. Applications of Kalman filtering include spacecraft orbit determination [4], target tracking [1,3,5], vehicle guidance and navigation [7,9,13,16,20], geophysical subsurface estimation [19], demographic modeling [15], and industrial process control [22,23].

The solution of the Kalman filter equations is expensive, in general requiring $O(n^3)$ operations for each state update, where n is the number of state variables. This has motivated the design of a number of parallel algorithms (see, e.g., [14] and the references contained therein). The majority of the proposed algorithms are targeted for implementation on special-purpose VLSI architectures such as systolic arrays and wavefront arrays. The reason is that the computations occurring in the Kalman filter are mostly matrix operations which can be efficiently mapped on systolic arrays.

On the other hand, little work has been done to demonstrate the feasibility of implementing the Kalman filter on commercially available parallel computers. To date, the only work we are aware of is by Baheti, Itzkowitz and O'Hallaron [10,17] who developed and implemented a parallel Kalman filter on a Warp computer. The Warp is a linear array of 10 or more identical and programmable cells connected to a general-purpose host computer. Baheti *et al.* designed a parallel algorithm for solving an n-state, m-measurement square root Kalman filter using an $(n+m+1)$-cell linear array. Based on this algorithm, they demonstrated on the Warp a mapping of a 9-state extended square root Kalman filter commonly used in target tracking applications.

This paper describes a parallel algorithm for square root Kalman filtering on a two-dimensional SIMD array and its implementation on the Connection Machine (CM). Manufactured by Thinking Machines Corporation, the CM is a SIMD architecture that consists of up to 64K data processors physically connected via a hypercube interconnection network [21].

The results of our investigation demonstrate that the square root Kalman filter can be efficiently mapped on the CM. Specifically, the CM has primitive *parallel prefix* or *scan* operations that facilitate the derivation of a simple, yet efficient, two-dimensional array algorithm. Moreover, using the CM's "virtual processing" facility, the same algorithm is applicable to filters of different sizes. Performance measurements show that the CM filter runs in time linear in the state vector size. This represents a great improvement over serial implementations which run in cubic time. Moreover, the CM is well suited for Kalman filter applications with large state vector sizes. Examples are satellite estimation, which may use between 40 to 200 states and missile guidance, which may use 10 to 20 states plus many bias variables.

We also investigated the applicability of the CM to large-scale applications such as multiple target tracking. In such an application, up to a hundred targets (e.g., satellites, aircrafts and missiles) may have to be tracked simultaneously, each requiring 3 or more different filters. Thus, up to 300 different Kalman filters may have to be processed simultaneously. For this application, we developed two mappings on the CM. The first mapping is based on the serial algorithm and assigns one processor per filter. The second mapping is based on the parallel algorithm and assigns a two-dimensional subarray of processors per filter. Both mappings achieved constant run-times, independent of the number of filters simultaneously processed.

The rest of the paper is organized as follows. Section 2 briefly describes the square root Kalman filter and its serial implementation. The most compute-intensive step in the solution of the filter equations is matrix

triangularization. Section 2 discusses how the triangularization problem can be solved using fast Givens rotations. Section 3 gives an overview of the CM and its scan operations. Section 4 focusses on the parallelization of the triangularization algorithm using the CM scan operations. For lack of space, the other steps of the parallel Kalman filter algorithm are only sketched (details can be found in the full paper [18]). The performance measurements are summarized in Section 5. Finally, Section 6 is devoted to conclusions and recommendations for further study.

2. THE SQUARE ROOT KALMAN FILTER

2.1. Mathematical Formulation

Consider a discrete-time space model given by

$$\mathbf{x}_{t+1} = \Phi_t \mathbf{x}_t + G_t \mathbf{w}_t \tag{1}$$

$$\mathbf{y}_t = H_t \mathbf{x}_t + \mathbf{v}_t$$

where the subscript t is the discrete time, $\mathbf{x}_t$ is an $(n \times 1)$ state vector, and $\mathbf{y}_t$ is an $(m \times 1)$ measurement vector (typically $m \leq n$). The process noise $\mathbf{w}_t$ and the measurement noise $\mathbf{v}_t$ are zero mean Gaussian random sequences. Φ_t, G_t, and H_t are time varying matrices of dimension $(n \times n)$, $(n \times n)$, and $(m \times n)$, respectively. It is assumed that $E(\mathbf{w}_t \mathbf{w}_\tau^T) = Q_t \delta_{t\tau}$, $E(\mathbf{v}_t \mathbf{v}_\tau^T) = R_t \delta_{t\tau}$, and $E(\mathbf{w}_t \mathbf{v}_\tau^T) = 0$ ($\delta_{t\tau}$ is the Kronecker delta).

The Kalman filter [12] gives an estimate $\hat{\mathbf{x}}_{t+1}$ of the state at time $t+1$ that is a linear combination of an estimate $\hat{\mathbf{x}}_t$ and the measurement data $\mathbf{y}_t$, at time t. More precisely, the predicted state estimate is given by the recursive relation

$$\hat{\mathbf{x}}_{t+1} = \Phi_t \hat{\mathbf{x}}_t + K_t[\mathbf{y}_t - H_t \hat{\mathbf{x}}_t]. \tag{2}$$

where:

$$K_t = \Phi_t P_t H_t^T R_{e,t}^{-1}, \tag{3}$$

$$R_{e,t} = H_t P_t H_t^T + R_t, \text{ and} \tag{4}$$

$$P_{t+1} = \Phi_t P_t \Phi_t^T + G_t Q_t G_t^T - K_t R_{e,t} K_t^T. \tag{5}$$

K_t is the Kalman gain matrix with dimension $(n \times m)$, $R_{e,t}$ is the covariance of the innovations with dimension $(m \times m)$, and P_t is the $(n \times n)$ covariance matrix.

2.2. The Kalman Filter - Square Root Formulation

It is well-known that the computation of matrix products of the form XYX^T (such as those in Equations 4 and 5) results in loss of information unless the calculations are done in double-precision. To avoid this, various matrix factorization methods (e.g., QR, LDL^T, etc.) are commonly employed. Based on earlier work by Kailath [11], Itzkowitz and Baheti [10] presented an algorithm for solving the Kalman filter equations by maintaining the matrices P_t and $R_{e,t}$ in factorized form:

$$P_t = L_{p,t} D_{p,t} L_{p,t}^T, \tag{6}$$

$$R_{e,t} = L_{e,t} D_{e,t} L_{e,t}^T,$$

where $L_{p,t}$ and $L_{e,t}$ are unit lower triangular matrices, and $D_{p,t}$ and $D_{e,t}$ are diagonal matrices. The Kalman gain matrix K_t and the covariance update P_{t+1} are then obtained from the triangularization of a particular matrix.

Let S be the $(m+n) \times (m+2n)$ matrix given by

$$S = \begin{bmatrix} R_{e,t} & H_t P_t \Phi_t^T \\ \Phi_t P_t H_t^T & \Phi_t P_t \Phi_t^T + G_t Q_t G_t^T \end{bmatrix}. \tag{7}$$

It was shown in [10] that S can be factored as

$$\begin{aligned} S &= ADA^T \\ &= \begin{bmatrix} I & H_t L_{p,t} & 0 \\ 0 & \Phi_t L_{p,t} & G_t \end{bmatrix} \begin{bmatrix} R_t & 0 & 0 \\ 0 & D_{p,t} & 0 \\ 0 & 0 & Q_t \end{bmatrix} \begin{bmatrix} I & 0 \\ L_{p,t}^T H_t^T & L_{p,t}^T \Phi_t^T \\ 0 & G_t^T \end{bmatrix} \end{aligned} \tag{8}$$

where A is an $(m+n) \times (m+2n)$ matrix and D is a $(m+2n) \times (m+2n)$ matrix. Under the assumption that R_t and Q_t are diagonal, D is also diagonal. Similarly, S can also be factored as

$$\begin{aligned} S &= A'D'(A')^T \\ &= \begin{bmatrix} L_{e,t} & 0 & 0 \\ K_t L_{e,t} & L_{p,t+1} & 0 \end{bmatrix} \begin{bmatrix} D_{e,t} & 0 & 0 \\ 0 & D_{p,t+1} & 0 \\ 0 & 0 & D_a \end{bmatrix} \begin{bmatrix} L_{e,t}^T & L_{e,t}^T K_t^T \\ 0 & L_{p,t+1}^T \\ 0 & 0 \end{bmatrix} \end{aligned} \tag{9}$$

where A' is an $(m+n) \times (m+2n)$ unit lower triangular matrix and D' is a $(m+2n) \times (m+2n)$ matrix. Because of the nature of this factorization, the matrix D_a is arbitrary.

The factorization of Equation 8 can be transformed into the factorization of Equation 9 using a triangularization algorithm based on fast Givens rotations. The triangularization algorithm is discussed in § 2.3.

Since the matrix A' contains $L_{p,t+1}$ and matrix D' contains $D_{p,t+1}$, the matrix P_{t+1} can be explicitly computed as soon as the triangularization procedure completes. However, this is not necessary since the recursive algorithm requires only the factorized version of P_{t+1} (see Equation 8).

The square root Kalman filter algorithm is presented as Algorithm K below. Assumed as given are an initial state estimate $\hat{\mathbf{x}}_0$ and an initial factored covariance matrix $P_0 = L_{p,0} D_{p,0} L_{p,0}^T$. Algorithm K is slightly more efficient than the algorithm described in [10]. Specifically, in [10] the term $K_t[\mathbf{y}_t - H_t \hat{\mathbf{x}}_t]$ was obtained by first computing K_t explicitly from $K_t = [K_t L_{e,t}][L_{e,t}]^{-1}$, then multiplying it with $\mathbf{c}_t = \mathbf{y}_t - H_t \hat{\mathbf{x}}_t$. (The matrices $[K_t L_{e,t}]$ and $[L_{e,t}]$ are obtained directly from matrix A'.) Excluding the time to compute $\mathbf{c}_t$, this method requires a total of $O(m^3 + nm^2 + nm)$ operations. On the other hand, step (3) of Algorithm K computes the same term without precomputing K_t, and instead uses forward substitution followed by a matrix-vector multiplication. Consequently, the number of operations is reduced to $O(m^2 + nm)$.

2.3. Matrix Triangularization Using Fast Givens Rotations

Let $A = [a_{ij}]$ be an $(M \times N)$ matrix with $M \leq N$, and $D = \text{diag}(d_1, ..., d_N)$ be an $(N \times N)$ diagonal matrix. We wish to transform ~~the pair~~ the pair of matrices (A, D) into another pair (A', D') such that: (1) A' is unit lower triangular and D' is diagonal, and (2) $ADA^T = A'D'(A')^T$. The desired transformation can be achieved using fast Givens rotations as described below.

Algorithm K:

Input: $\hat{\mathbf{x}}_0$, $L_{p,0}$, $D_{p,0}$; Φ_t, H_t, R_t, Q_t, G_t, $\mathbf{y}_t$, for $t \geq 0$.

Output: $\hat{\mathbf{x}}_t$ for $t > 0$.

Steps: For $t = 0, 1, 2, \cdots$, do the following:

(1) Set up matrices A and D such that:

$$A = \begin{bmatrix} I & H_t L_{p,t} & 0 \\ 0 & \Phi_t L_{p,t} & G_t \end{bmatrix} \quad \text{and} \quad D = \begin{bmatrix} R_t & 0 & 0 \\ 0 & D_{p,t} & 0 \\ 0 & 0 & Q_t \end{bmatrix}.$$

(2) Triangularize A to obtain:

$$A' = \begin{bmatrix} L_{e,t} & 0 & 0 \\ K_t L_{e,t} & L_{p,t+1} & 0 \end{bmatrix} \quad \text{and} \quad D' = \begin{bmatrix} D_{e,t} & 0 & 0 \\ 0 & D_{p,t+1} & 0 \\ 0 & 0 & D_a \end{bmatrix},$$

such that $S = ADA^T = A'D'(A')^T$.

(3) Compute $K_t[\mathbf{y}_t - H_t\hat{\mathbf{x}}_t]$ as follows:
(a) Compute $\mathbf{c}_t = \mathbf{y}_t - H_t\hat{\mathbf{x}}_t$.
(b) Solve $L_{e,t}\mathbf{z}_t = \mathbf{c}_t$ for $\mathbf{z}_t$.
(c) Compute $K_t[\mathbf{y}_t - H_t\hat{\mathbf{x}}_t] = [K_t L_{e,t}]\mathbf{z}_t$.

(4) Compute and output: $\hat{\mathbf{x}}_{t+1} = \Phi_t\hat{\mathbf{x}}_t + K_t[\mathbf{y}_t - H_t\hat{\mathbf{x}}_t]$.

For $1 \leq k \leq M$, define T_{kk} as the function that transforms (A, D) into (B, C) where:

(1) C is identical to D except for k-th diagonal element which is given by $c_k = a_{kk}^2 d_k$.

(2) B is identical to A except for the elements in column k which are given by $b_{ik} = a_{ik}/a_{kk}$.

It can be shown that $ADA^T = BCB^T$. Moreover, $b_{kk} = 1$.

Similarly, for $1 \leq k \leq M$ and $k < j \leq N$, define $T_{kj}(A, D) = (E, F)$ where:

(1) F is identical to D except for the k-th and j-th diagonal elements which are given by $f_k = a_{kk}^2 d_k + a_{kj}^2 d_j$ and $f_j = d_k d_j / f_k$.

(2) E is identical to A except for the elements in columns k and j which are given by $e_{ik} = (a_{kk} a_{ik} d_k + a_{kj} a_{ij} d_j)/f_k$ and $e_{ij} = a_{kk} a_{ij} - a_{kj} a_{ik}$.

Again, it can be shown that $ADA^T = EFE^T$. Also, $e_{kj} = 0$.

Thus, to triangularize matrix A, we zero out the rows of A one at a time beginning with the first row. For row k, the diagonal element a_{kk} is set to unity by applying T_{kk}, then the elements $\{a_{kj} \mid k+1 \leq j \leq N\}$ are

set to zero in turn by applying $T_{k,k+1}$, $T_{k,k+2}$, ..., T_{kN}.

A serial implementation of Algorithm K requires approximately $4.3n^3 + 11.5n^2m + 10nm^2$ scalar operations per state update [18]. The matrix triangularization step alone (step (2)) takes up roughly three-fourths of this total time ($3.3n^3 + 4n^2m + 2nm^2$ to be precise). Consequently, for the CM implementation, we focussed on developing an efficient parallel algorithm for matrix triangularization. The parallel algorithm makes efficient use of *scan* operations, which are available as primitive instructions on the CM. A brief description of these instructions is given in the next section.

3. OVERVIEW OF THE CONNECTION MACHINE

The *Connection Machine (CM)* [21] is a parallel SIMD architecture consisting of up to 64K processing elements called *data processors*. The data processors are collectively under the control of a single control unit called the *sequencer*. The task of the sequencer is to decode instructions issued by a front end computer and broadcast them to the data processors, which then all execute the same instruction simultaneously.

The CM data processors are physically connected via a hypercube interconnection network; however, this physical interconnection is transparent to the user. The CM provides a facility called "virtual processing" which allows programs to be written assuming *any* appropriate number of virtual processors; the mapping from virtual processors to physical processors in done automatically by the CM. A collection of virtual processors is called a *virtual processor set*, or *VP set*. Associated with a VP set is a *geometry* which specifies the logical interconnection of the virtual processors. Geometries currently supported by the CM are d-dimensional grids, where d is any number between 1 and 31 inclusive. The CM has primitive instructions for nearest-neighbor communication within the VP set (called NEWS communication). In addition, general communication instructions are provided for data transfer between non-adjacent processors. NEWS communication is faster than general communication; the CM performs virtual-to-physical mapping such that any two nearest-neighbor virtual processors are either assigned to the same physical processor or to

The third class of communication operations are called *parallel prefix* or *scan* operations. These differ from the first two classes in that while communicating, the virtual processors also perform some combining operation (such as addition) on the data they receive. Informally, a scan operation takes a binary associative operator $\circ$ and an ordered set of elements $[\, a_1, a_2, a_3, \cdots \,]$, and computes the ordered set $[\, a_1, (a_1 \circ a_2), (a_1 \circ a_2 \circ a_3), \cdots \,]$.

For example, for a scan-with-add operation, the computed values are the prefix sums $\sum_{k=1}^{i} a_k$ for all i. The ordered set of elements typically comes from an ordered sequence of processors along a specific axis of the given VP set.

In general, the scan instruction has two attributes: (1) *direction*, which is either upward or downward and (2) *inclusion*, which is either inclusive or exclusive. The direction indicates the ordering of the values along the axis: upward (downward) orders the values according to increasing (decreasing) processor NEWS coordinates along the axis. The inclusion indicates whether or not the value originally stored in a processor will be included in the partial sum computed for that processor. To illustrate the effect of these attributes on a scan-with-add operation, consider a linear array of N processors such that processor i initially contains a value $v(i)$. If an upward, inclusive, scan-with-add were performed, then processor i would get the value $\sum_{k=1}^{i} v(k)$.

If instead the inclusion were exclusive, then processor i would get the value $\sum_{k=1}^{i-1} v(k)$ and processor 1 would get the value 0. For a downward, inclusive, scan-with-add, processor i would get $\sum_{k=N}^{i} v(k)$; if the inclusion were exclusive, then processor i would get $\sum_{k=N}^{i+1} v(k)$ and processor N would get the value 0.

A special case of the scan operation is the *reduce* operation. A reduce-wth-add computes only the total sum $\sum_{k=1}^{N} v(k)$; this total sum is then stored in a specific destination processor which is also an argument to the reduce operation (all other processors are unaffected). Another useful instruction is *spread-with-copy*, which causes data from a specified source processor to be broadcast to all processors (including the source processor itself).

4. PARALLEL SQUARE ROOT KALMAN FILTER ON THE CONNECTION MACHINE

4.1. Matrix Triangularization Using Scan Operations

Consider again step (2) of Algorithm K which transforms the pair of matrices (A, D) into another pair (A', D') such that A' is unit lower triangular. In § 2.3 it was shown that the required transformation can be achieved using fast Givens rotations as follows (recall that A is an $(M \times N)$ matrix and D is an $(N \times N)$ diagonal matrix, where $M = m+n$ and $N = m+2n$):

for $k = 1$ **to** M **do**
  for $j = k$ **to** N **do**
    $(A, D) = T_{kj}(A, D)$;
  endfor;
endfor.

For $1 \le k \le M$, let $A^{(k)}$ and $D^{(k)}$ be the matrices A and D, respectively, at the end of iteration k of the outer loop. Similarly, let $A^{(0)}$ and $D^{(0)}$ be the original matrices. The idea behind the parallel triangularization algorithm is to parallelize the inner loop, i.e., to compute $A^{(k)}$ and $D^{(k)}$ directly from $A^{(k-1)}$ and $D^{(k-1)}$ using a constant number of scan operations.

For $k \le i \le M$ and $k \le j \le N$, define

$$s_{ij}^{(k-1)} = \sum_{l=k}^{j} a_{kl}^{(k-1)} a_{il}^{(k-1)} d_l^{(k-1)}. \tag{10}$$

In [18] we showed that the elements of $A^{(k)}$ and $D^{(k)}$ can be expressed succinctly in terms of the partial sums in Equation 10. More precisely, the first $(k-1)$ diagonal elements of $D^{(k)}$ have the same values as those of $D^{(k-1)}$; the remaining elements have new values:

$$d_k^{(k)} = s_{kN}^{(k-1)}, \text{ and} \tag{11}$$

$$d_j^{(k)} = \frac{s_{k,j-1}^{(k-1)}}{s_{kj}^{(k-1)}} d_j^{(k-1)}, \; k < j \le N.$$

Similarly, the first $(k-1)$ rows and first $(k-1)$ columns of $A^{(k)}$ have the same values as those of $A^{(k-1)}$; the remaining elements have new values:

$$a_{ik}^{(k)} = \frac{s_{iN}^{(k-1)}}{s_{kN}^{(k-1)}}, \text{ and} \tag{12}$$

$$a_{ij}^{(k)} = a_{ij}^{(k-1)} - \frac{s_{i,j-1}^{(k-1)}}{s_{k,j-1}^{(k-1)}} a_{kj}^{(k-1)}, \; k \le i \le M \text{ and } k < j \le N.$$

Equations 10 to 12 suggests a parallel algorithm that maps nicely on the CM. For an $(M \times N)$ matrix A and an $(N \times N)$ diagonal matrix D, the parallel algorithm uses a two-dimensional VP set P of dimension $M \times (N+1)$. The VP set has two memory fields, a and d, for storing the elements of matrices A and D, respectively. Initially, $a(i,j) = a_{ij}$ and $d(i,j) = d_j$ for $1 \le i \le M$ and $1 \le j \le N$. The idea behind the parallel algorithm is simple. For each iteration k, $1 \le k \le M$, the partial sums $\{s_{ij}^{(k-1)}\}$ are computed in parallel using scan-with-add operations. The partial sums are distributed to the right processors, which then update the matrix elements in their a and d fields. The steps are given in Algorithm PT below. The triangularization is done in place; i.e., if A' is the triangularized matrix and D' is the corresponding diagonal matrix, then at the end of the algortihm, $a(i,j) = a'_{ij}$ and $d(M,j) = d'_j$ for $1 \le i \le M$ and $1 \le j \le N$.

Algorithm PT:

For k = 1 to M do the following:

(1) For all columns j, $k \le j \le N$, spread-with-copy $a(k,j)$ down the column and store it in $b(i,j)$, $k \le i \le M$. (Thus, $b(i,j) = a_{kj}^{(k-1)}$.)

(2) For $k \le i \le M$ and $k \le j \le N$, compute $t_1(i,j) = a(i,j) \cdot b(i,j) \cdot d(i,j)$. (Thus, $t_1(i,j) = a_{ij}^{(k-1)} a_{kj}^{(k-1)} d_j^{(k-1)}$.)

(3) For all rows i, $k \le i \le M$, perform an upward, exclusive, scan-with-add over the set of values $\{t_1(i,j) \mid k \le j \le N+1\}$. (Thus, $t_1(i,j) = s_{j-1}^{(k-1)}$.)

(4) For all columns j, $k \le j \le N$, spread-with-copy $t_1(k,j)$ down the column and store it in $t_2(i,j)$, $k \le i \le M$. (Thus, $t_2(i,j) = s_{j-1}^{(k-1)}$.)

(5) For $k \le i \le M$ and $k < j \le N+1$, compute $t_1(i,j) = t_1(i,j)/t_2(i,j)$. (Thus, $t_1(i,j) = s_{j-1}^{(k-1)} / s_{j-1}^{(k-1)}$; in particular, $t_1(i,N+1) = a_{ik}^{(k)}$ and $t_2(i,N+1) = d_k^{(k)}$.)

(6) For $k \le i \le M$, send $t_1(i,N+1)$ to $a(i,k)$ and send $t_2(i,N+1)$ to $d(i,k)$. (Thus, $a(i,k) = a_{ik}^{(k)}$ and $d(i,k) = d_k^{(k)}$.)

(7) For $k \le i \le M$ and $k < j \le N$, compute $a(i,j) = a(i,j) - b(i,j) \cdot t_1(i,j)$. (Thus, $a(i,j) = a_{ij}^{(k)}$.)

(8) For $k \le i \le M$ and $k < j \le N$, get $t_1(i,j)$ from $t_2(i,j+1)$. (Thus, $t_1(i,j) = s_{kj}^{(k-1)}$ and $t_2(i,j) = s_{k,j-1}^{(k-1)}$.)

(9) For $k \le i \le M$ and $k < j \le N$, compute $d(i,j) = d(i,j) \cdot t_2(i,j)/t_1(i,j)$. (Thus, $d(i,j) = d_j^{(k)}$.)

4.2. Parallel Implementation of Algorithm K

Steps (1), (3) and (4) of Algorithm K can also be parallelized using the CM scan operations. For example, consider the computation of the matrix products $H_t L_{p,t}$ and $\Phi_t L_{p,t}$ in step (1). Here, Φ_t and $L_{p,t}$ are $(n \times n)$ matrices and H_t is an $(m \times n)$ matrix. Both matrix products can be computed in n iterations as

follows. Initially, H_t (Φ_t) is stored in the same set of processors where $H_t L_{p,t}$ ($\Phi_t L_{p,t}$) will eventually reside. $L_{p,t}$ (obtained from the previous triangularization step) is stored in the same place where Φ_t is (but in a different memory field). The elements of the matrix products will be stored in a separate field, say f, which is initially zero for all processors. (Only the processors containing H_t and Φ_t will participate in the ensuing computation.) At each iteration k, $1 \le k \le n$, the k-th row of $L_{p,t}$ is broadcast to all the rows and the k-th column of the composite matrix $[H_t \mid \Phi_t]^T$ is broadcast to all the columns. The two elements received by each processor are then multiplied and added to the current contents of field f. It is easy to verify that at the end of n iterations, field f will contain the desired matrix products.

Computing the matrix-vector products $H_t \hat{\mathbf{x}}_t$ in substep (3a) and $\Phi_t \hat{\mathbf{x}}_t$ in step (4) can be accomplished with a single reduce-with-add operation. First, $\hat{\mathbf{x}}_t$ (again, computed in the previous iteration) is broadcast to all rows of the composite matrix $[H_t \mid \Phi_t]^T$ and then multiplied element-wise with the each row. A reduce-with-add operation is then performed on each row to obtain the desired matrix-vector products.

Substeps (3b) and (3c) can actually be performed *while* the first m rows of matrix A are being triangularized. Let $\theta = [\theta_1, ..., \theta_{m+n}]^T$ be the $(m+n) \times 1$ vector defined by

$$\theta_i = c_t^{(i)} - \sum_{k=1}^{i-1} L_{e,t}^{(i,k)} \cdot \theta_k, \quad 1 \le i \le m, \text{ and} \tag{13}$$

$$\theta_{i+m} = -\sum_{k=1}^{m} [K_t L_{e,t}]^{(i,k)} \cdot \theta_k, \quad 1 \le i \le n,$$

where $L_{e,t}^{(i,k)}$ and $[K_t L_{e,t}]^{(i,k)}$ denote the (i,k)-th element of $L_{e,t}$ and $[K_t L_{e,t}]$, respectively, and $c_t^{(i)}$ denotes the i-th element of $\mathbf{c}_t$. It can be verified that the first m elements of θ is the $(m \times 1)$ vector $\mathbf{z}_t$ computed in substep (3b). Similarly, the last n elements of θ is $-[K_t L_{e,t}]\mathbf{z}_t$, the negative of the $(n \times 1)$ vector computed in substep (3c). Now, in the triangularization algorithm, the updated columns of the matrix A are always first computed at rightmost column of the VP set (see step (5) of Algortihm *PT*) before they are routed to their final destinations. Thus, for the first m iterations, columns 1, 2, ..., m of the composite matrix $[L_{e,t} \mid K_t L_{e,t}]^T$ appear, in turn, at the rightmost column of the VP set. By exploiting this fact, the vector θ can be computed in m iterations as follows. First, θ is positioned in the rightmost column of the VP set and initialized so that its first m elements comprise the vector $\mathbf{c}_t$ and its last n elements are zero. At iteration k, the k-th element of θ is broadcast down the column and multiplied with the elements of the composite matrix $[L_{e,t} \mid K_t L_{e,t}]^T$ that arrive during this iteration. These computed values represent the terms $L_{e,t}^{(i,k)} \cdot \theta_k$ and $[K_t L_{e,t}]^{(i,k)} \cdot \theta_k$ in Equation (13). These values are then subtracted from the corresponding elements of θ and the next iteration is ready to begin. At the end of m iterations, θ would have been computed; the last n elements are negated to obtain the matrix product $[K_t L_{e,t}]\mathbf{z}_t = K_t \mathbf{c}_t$.

We have some final remarks. In the actual CM implementation, we exploited the sparsity of the matrix A to reduce the size of the VP set from $(m+n) \times (m+2n+1)$ to $(m+n) \times (n+1)$. This was done by storing only columns $m+1$ through $m+n$ of the matrix A (see step (1) of Algorithm *K*). The nonzero matrix containing G_t was stored in the same set of processors holding $\Phi_t L_{p,t}$ but with the rows reversed. (Imagine folding matrix A along the vertical line separating $\Phi_t L_{p,t}$ from G_t.) The triangularization step is essentially the same as before except that the partial sums computed in Equation (10) are slightly modified to take into account the contribution due to the now missing first m columns of A (which contains $[I \mid 0]^T$). Also, when zeroing out the last n rows of A, the row-scan operation (step (3) of Algortihm *PT*) is first performed towards the right (as before), then towards the left to pick up the contribution due to G_t. Note also that the matrices $L_{e,t}$ and $[K_t L_{e,t}]$ are no longer stored (they are supposed to lie in the first m columns of A); this creates no problem

since they are used only in computing the vector θ as described in the previous paragraph.

4.3. Complexity Analysis of The Parallel Implementation

Table 1 lists the numbers of CM instructions of various types required by the parallel square root Kalman filter algorithm. These instruction counts are based on our actual CM implementation which used the reduced VP set. We have included row-reduce-with-add in the line labelled row-scan-with-add. A separate column gives the instruction counts for each step of Algorithm K.

Instruction Counts for Parallel Implementation of Algorithm K					
Operation	Step (1)	Step (2)	Step (3)	Step (4)	Total
add/sub	n	$3n+2m$	$m+2$	1	$4n+3m+3$
mult/div	n	$12n+6m$	$m+1$	1	$13n+7m+2$
row-scan-with-add	0	$2n+m$	1	1	$2n+m+2$
col-spread-with-copy	$n+1$	$4n+2m$	$m+1$	1	$5n+3m+3$
NEWS comm.	0	$6n+m$	0	0	$6n+m$
gen. comm.	0	0	0	1	1

Table 1.

As Table 1 indicates, the total number of CM instructions executed is linear in n (assuming $n \geq m$). We should point out, however, that the "real" complexity of the parallel algorithm is $O(n\log(n))$ rather than $O(n)$. The reason is that the CM scan instructions are in reality *not* constant-time operations. For instance, a row-scan-with-add, which operates on a row of size $n+1$, actually takes $\log(n+1)$ unit routing steps and $\log(n+1)$ additions on the hypercube, the underlying network of the CM. Similarly, a column-spread-with-copy on a column of size $m+n$ requires $\log(m+n)$ unit routing steps.

Table 1 also allows us to determine, at least theoretically, the fractions of the total time due to communication and computation. In Table 1, the first two rows contribute only to the computation time while the last three rows contribute only to the communication time. For the row-scan-with-add, half of the time is due to communication and the other half is due to computation. Under the assumption that one unit routing step takes as much time as one arithmetic operation, then communication will be about 78% of the total time.

5. PERFORMANCE RESULTS

5.1. Performance Results for Different Filter Sizes

The parallel square root Kalman filter was implemented on an 8K Connection Machine with floating-point hardware. In addition, the serial algorithm (Algorithm K) was run on two types of serial processors: (1) a Sun 3/60 computer with a 68881 floating-point coprocessor and (2) a *single* CM data processor. For each

implementation, the run-time per state update was measured for state vector sizes and measurement vector sizes between 2 and 40 inclusive, at increments of 2.

Figure 1 gives a plot of the run-times for the three implementations as functions of the state vector size (with the measurement vector size equal to the state vector size). From the graphs, it can be seen that the run-times for the two serial implementations (i.e., on the single data processor and on the Sun) are roughly cubic functions of the state vector size n. These agree with the complexity analysis of the serial algorithm. For the parallel implementation, the run-time is nearly linear in n (a rough estimate is $21n + 30$). This is quite better than the $O(nlog(n))$ run-time derived from the complexity analysis of the parallel algorithm. We believe that the $O(n\log(n))$ behavior of the run-time will become noticeable only for very large n.

Observe from the graphs that the parallel CM filter beats the serial filter on a single CM data processor for $n \geq 5$ but beats the Sun filter only for larger state vector sizes, i.e., $n \geq 32$. As elaborated in § 6, the reason is that a single data processor has only a fraction of the computing speed of a Sun processor. For a SIMD architecture with more powerful processors than the CM, one would expect that the parallel algorithm will become faster than Sun filter at a lower value of n.

Our parallel complexity analysis indicates that communication time will dominate computation time in the parallel implementation. This was also verified by the performance results, as shown in Figure 2. The graph plots the communication time, computation time, and total time as a function of n, with $m = n$. As can be seen from the graph, the run-time is roughly 70% communication and 30% computation.

5.2. Performance Results for Multiple Filters/Targets

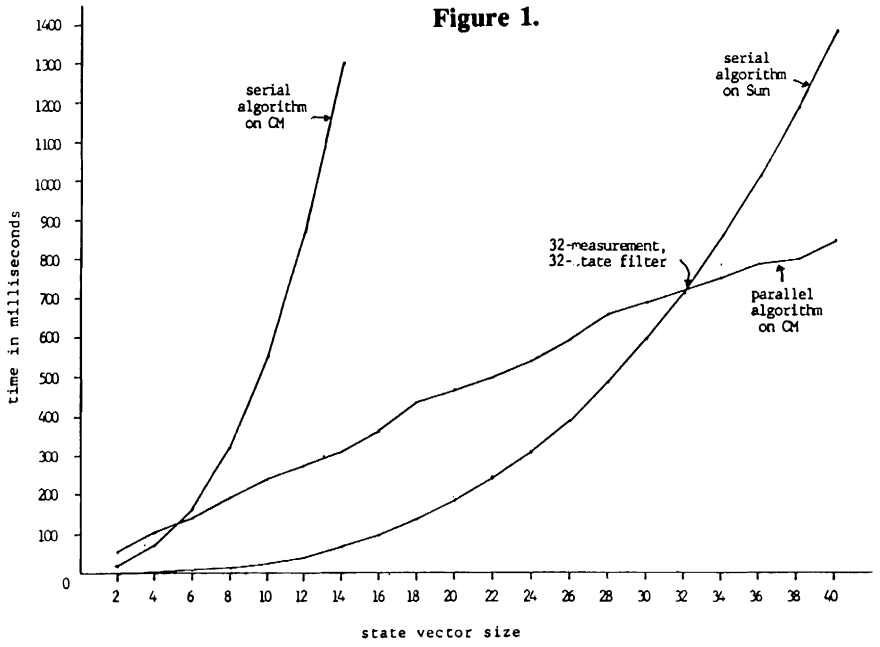
We also considered a specific multiple target tracking application, using the 9-state, 3-measurement square root extended Kalman filter described by Baheti and his colleagues [10,17]. For this application, a large number of targets are to be tracked simultaneously, each using a separate filter. The tracking filter is based on the laws of motion and a stochastic acceleration model. The (9×1) state vector $\mathbf{x}_t = [x, y, z, \dot{x}, \dot{y}, \dot{z}, \ddot{x}, \ddot{y}, \ddot{z}]^T$ gives the position, velocity, and acceleration of the target in the x, y, and z axes. The (3×1) measurement vector $\mathbf{y}_t = [r, \theta_T, \theta_E]^T$ contains the noisy measurements of the range, azimuth, and elevation angles, respectively. The discrete-time equation of target motion is similar to Equation 1 (§ 2.1), except that the term $H_t\mathbf{x}_t$ is replaced by the (3×1) vector $\mathbf{h}(\mathbf{x}_t)$ which transforms the Cartesian coordinates in $\mathbf{x}_t$ to polar coordinates (a nonlinear transformation).

The extended Kalman filter uses linearization of the nonlinear equations about current state estimates. The structure of the covariance and gain computation equations in the extended Kalman filter are similar to Equations 2 to 5 (2.1) except that: (1) in Equation 2 the term $H_t\hat{\mathbf{x}}_t$ is replaced by the coordinate transformation vector $\mathbf{h}(\hat{\mathbf{x}}_t)$, and (2) in all equations the matrix H_t is used to denote the Jacobian matrix of $\mathbf{h}(\mathbf{x})$ evaluated at $\hat{\mathbf{x}}_t$. Thus, the extended Kalman filter algorithm is the same as Algorithm K except that in addition, the coordinate transformation vector $\mathbf{h}(\hat{\mathbf{x}}_t)$ and the Jacobian matrix H_t are computed at each update step.

We developed and implemented two different mappings of the extended Kalman filter (for multiple targets) on the CM. The first mapping assigns a single CM data processor to each filter. The data processor solves the filter equations concurrently with, and independently of, other data processors. The second mapping is based on the parallel Kalman filter algorithm: each filter is assigned a (12×10) subarray of CM data processors to solve the equations in data parallel fashion. The advantage of the first mapping is that it requires no inter-processor communciation. However, because the filter equations are solved serially, the number of computation steps is larger than the second mapping. The serial extended Kalman filter algorithm was also

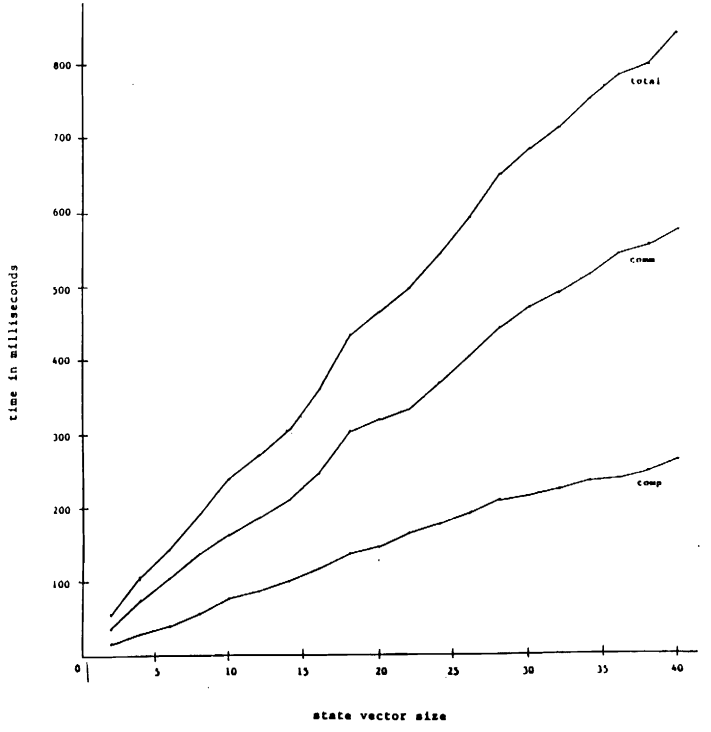
SINGLE KALMAN FILTER RUNTIMES
VERSUS FILTER SIZE

Figure 1.



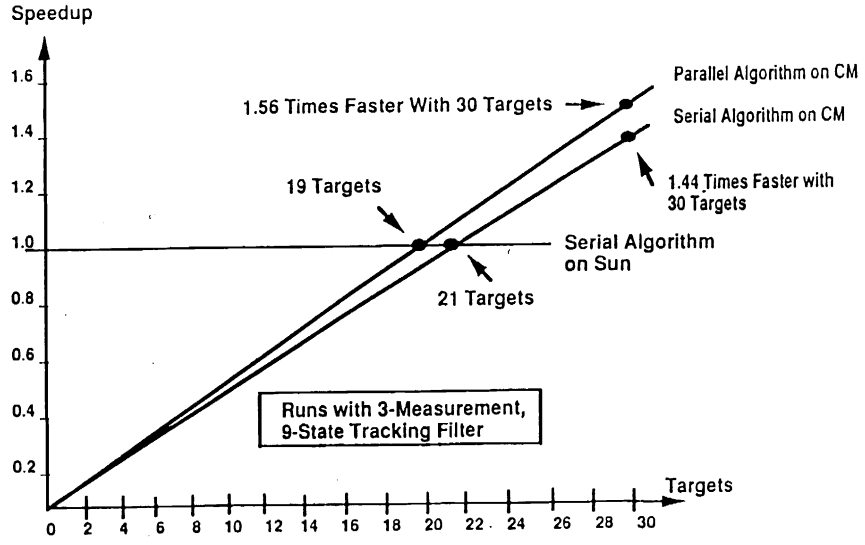
COMMUNICATION TIME AND COMPUTATION TIME
VERSUS FILTER SIZE

Figure 2.



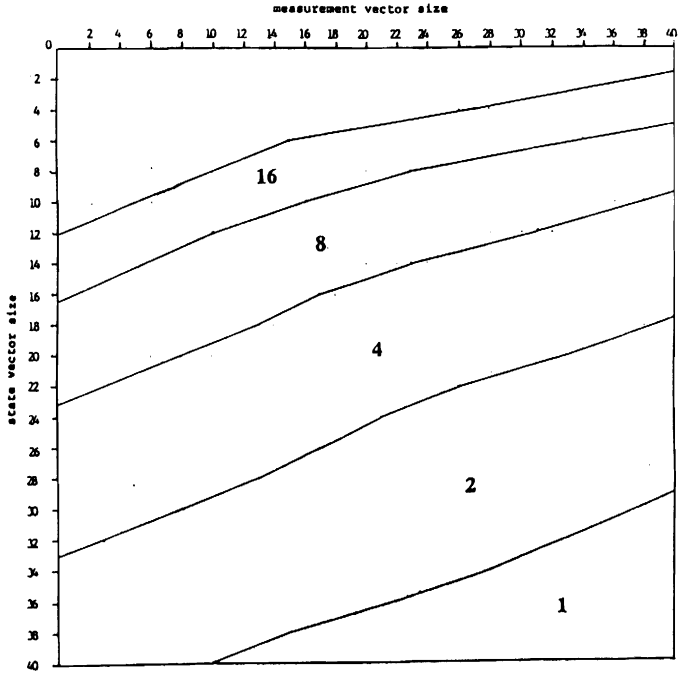
SPEEDUP OF CONNECTION MACHINE OVER SUN

Figure 3.



PARALLEL CM ALGORITHM VERSUS
SERIAL SUN ALGORITHM

Figure 4.



implemented on a Sun 3/60.

Both mappings share the desirable property that the time per state update remains constant regardless of the number of targets simultaneously tracked. In contrast, for a serial implementation the time per state update increases linearly with the number of targets. These were verified by our performance results. For the first mapping (one processor per filter), the run-time was roughly constant at 192 milliseconds for any number of targets. The second mapping ((12×10)-processor subarray per filter) was slightly faster at 178 milliseconds. On the other hand, the serial algorithm on the Sun took approximately $9.25n_t$ milliseconds, where n_t is the number of targets. Figure 3 shows a plot of the speedup for the two mappings over the serial Sun algorithm. As shown in the figure, the first mapping was faster than the Sun filter for 21 or more targets while the second mapping was faster than the Sun filter for 19 or more targets.

5.3. Combined Performance Results

The performance results described in the last two subsections can be combined to compare the performance of the parallel CM filter and the serial Sun filter when the filter size and the number of targets are varied simultaneously. This is shown in Figure 4. In the region marked X, the CM is faster than the Sun for X or more targets. For example, for a 10-state, 10-measurement filter, the CM is faster than the Sun if the number of targets is 16 or more. For a 20-state, 20-measurement filter, the CM is faster than the Sun for 4 or more targets. In general, the CM beats the Sun for less number of targets as the filter size increases.

6. CONCLUSIONS AND RECOMMENDATIONS FOR FURTHER STUDY

The results of our investigation demonstrate that the square root Kalman filter can be conveniently mapped to SIMD array architectures like the CM. On a two-dimensional SIMD array, the parallel Kalman filter algorithm achieves greater parallelism than the one-dimensional array algorithm reported in [10,17]. The CM, in particular, has primitive parallel prefix or scan operations which facilitate the derivation of a relatively simple, yet efficient, parallel algorithm. Moreover, because of the CM's scalability, the same algorithm is applicable to filters of different sizes.

The CM is best suited for Kalman filter applications that process large data sets. Examples are applications with large state vector sizes, e.g., satellite estimation, which may use 40 to 200 states, and missile guidance which may use 10 to 20 states plus many bias variables. Another application is multiple target tracking, in which up to 100 targets may have to be tracked simultaneously, each target requiring the use of 3 or more different Kalman filters. Thus, up to 300 different Kalman filters may have to processed simultaneously.

That the CM filter is superior to the Sun filter only for large filter sizes stems from the fact that, although the CM has a large number of data processors, each data processor has limited processing capability. In particular, computation and communication in the CM is bit-serial and hence quite slow. Even with floating-pont hardware, a CM data processor is still much slower than the Sun processor. The reason is that in the CM, 32 data processors share a single floating-point coprocessor, so that a parallel floating-point operation supposedly performed in parallel by 32 data processors is in fact done serially by the floating-point hardware. Clearly, the performance of the parallel Kalman filter algorithm will be superior on a SIMD array with better processing capability than the CM, e.g., one with powerful individual floating-point processors and bit-parallel communication. On such an architecture, one should expect the regions in Figure 4 to shift upwards towards the origin. That is, for the same filter size, the SIMD array implementation will become faster than the Sun

implementation for smaller number of targets.

Our preliminary investigation suggests directions for future study. The two mappings we developed for the CM represent two extremes of a range of possible mappings. The serial mapping assigns only one CM data processor to a filter, while the parallel mapping assigns a separate processor to each matrix element. Even though the parallel mapping spent about 70% of the time communicating and only 30% computing, it still ran faster than the serial mapping. We believe that even faster run-times can be obtained using a "hybrid" mapping that assigns small blocks of matrix elements to single processors but still distributes the matrix over multiple processors. This way, communication is reduced at the expense of greater computation. The key idea is to find the right partitioning that balances computation and communication.

There is also room for improvement in the parallel algorithm we developed for the CM. The presence of parallel prefix operations as primitive instructions in the CM was instrumental in deriving a relatively simple parallel algorithm. However, the theoretical time complexity of the parallel algorithm is $O(n\log(n))$ (where n is the state vector size), which is a logarithmic factor slower than proposed systolic algorithms. One interesting problem is to implement an $O(n)$ systolic algorithm on the CM [14]. This problem is not trivial because of the need to simulate the I/O-computation overlap frequently occurring in systolic algorithms.

Finally, we would like to see similar performance measurements for the Kalman filter on other commercially available architectures. Of immediate interest is extending the work of Baheti *et. al* in [10,17] on the 10-cell Warp computer. They reported performance measurements only for a single 9-state, 3-measurement extended Kalman filter. It would be nice to compare the performance of their Warp algorithm with our CM algorithm for different filter sizes and for multiple filters.

7. REFERENCES

[1] Aidala, V. J. and Hammel, S. E., Utilization of modified polar coordinates for bearings-only tracking, *IEEE Trans. Automat. Contr.*, vol. AC-28, pp. 283-294, Mar. 1983.

[2] Baheti, R. S., personal communication.

[3] Berg, R. F., Estimation and prediction for maneuvering target trajectories, *IEEE Trans. Automat. Contr.*, vol. AC-28, pp. 294-304, Mar. 1983.

[4] Campbell, J. K., Synnott, S. P. and Bierman, G. J., Voyager orbit determination at Jupiter, *IEEE Trans. Automat. Contr.*, vol. AC-28, pp. 256-268, Mar. 1983.

[5] Daum, F. E. and Fitzgerald, R. J., Decoupled Kalman filters for phased array radar tracking, *IEEE Trans. Automat. Contr.*, vol. AC-28, pp. 269-283, Mar. 1983.

[6] Friedland, B., Treatment of bias in recursive filtering, *IEEE Trans. Automat. Contr.*, vol. AC-14, pp. 359-367, Aug. 1969.

[7] Fung, P. T. and Grimble, M. J., Dynamic ship positioning using a self-tuning Kalman filter, *IEEE Trans. Automat. Contr.*, vol. AC-28, pp. 339-349, Mar. 1983.

[8] Greenberg, A. G. and Hajek, B., Deflection routing in hypercube networks, manuscript, 1989.

[9] Hostetler, L. D. and R. D. Andreas, Nonlinear Kalman filtering techniques for terrain-based navigation, *IEEE Trans. Automat. Contr.*, vol. AC-28, pp. 315-323, Mar. 1983.

[10] Itzkowitz, H. R. and Baheti, R. S., Demonstration of square root Kalman filter on Warp parallel computer, *Proceedings of the 1989 American Control Conference*.

[11] Kailath, T., State-space modelling: square root algorithms, in *Systems and Control Encyclopedia* (M. G. Singh, ed.), pp. 2618-2622, Pergamon Press, 1987.

[12] Kalman, R. E., A new approach to linear filtering and prediction problems, *Trans. ASME, J. Basic Eng.*, Series 82D, pp. 35-45, Mar. 1960.

[13] Kao, M. H. and Eller, D. H., Multi-configuration Kalman filter design for high-performance GPS navigation, *IEEE Trans. Automat. Contr.*, vol. AC-28, pp. 304-314, Mar. 1983.

[14] Kung, S. Y., *VLSI Array Processors*, Prentice-Hall, Englewood Cliffs, New Jersey, 1988, pp. 561-571.

[15] Liebundgut, B. G., Rault, A. and Gendreau, F., Application of Kalman filtering to demographic models, *IEEE Trans. Automat. Contr.*, vol. AC-28, pp. 427-434, Mar. 1983.

[16] Mealy, G. L. and Tang, W., Application of multiple model estimation for a recursive terrain height correlation system, *IEEE Trans. Automat. Contr.*, vol. AC-28, pp. 323-331, Mar. 1983.

[17] O'Hallaron, D. R. and Baheti, R. S., Fast mapping of a Kalman filter on Warp, *Proceedings of SPIE Symposium, Real-Time Signal Processing XI*, vol. 977, San Diego, CA, August 1988.

[18] Palis, M. A. and Krecker, D. K., Parallel Kalman Filtering on the Connection Machine, GE Internal Technical Report, Advanced Technology Laboratories, GE Aerospace, Moorestown, New Jersey, 1989.

[19] Ruckebusch, G., A Kalman filtering approach to natural gamma ray spectroscopy in well logging, *IEEE Trans. Automat. Contr.*, vol. AC-28, pp. 372-380, Mar. 1983.

[20] Saelid, S., Jenssen, N. A. and Balchen, J. G., Design and analysis of a dynamic positioning system based on Kalman filtering and optimal control, *IEEE Trans. Automat. Contr.*, vol. AC-28, pp. 331-338, Mar. 1983.

[21] Thinking Machines Corporation, *Connection Machine Model CM-2 Technical Summary*, May 1989.

[22] Tylee, J. L., On-line failure detection in nuclear power plant instrumentation, *IEEE Trans. Automat. Contr.*, vol. AC-28, pp. 406-415, Mar. 1983.

[23] Wallace, J. N. and Clarke, R., The application of Kalman filtering estimation techniques in power station control systems, *IEEE Trans. Automat. Contr.*, vol. AC-28, pp. 416-427, Mar. 1983.

--