Proceedings of the 8th WSEAS International Conference on APPLIED MATHEMATICS, Tenerife, Spain, December 16-18, 2005 (pp37-42)

An OpenMP+/MPI Recursive Least Squares pipelined Parallel Algorithm based on the square root and extended version Information Filter for heterogeneous parallel systems

F.J. MARTÍNEZ ZALDÍVAR[†], A.M. VIDAL MACIÁ[‡] and A. GONZÁLEZ SALVADOR[†]

[†]Departamento de Comunicaciones
[‡]Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Camino de Vera s/n, 46022 Valencia
SPAIN
fjmartin@dcom.upv.es, avidal@dsic.upv.es, agonzal@dcom.upv.es

Abstract: - This article describes a parallel OpenMP and/or MPI implementation of an algorithm for solving the Recursive Least Squares (RLS) problem, suitable for heterogeneous parallel systems. A model for automatic/adaptive load balancing is proposed for either homogeneous or heterogeneous parallel systems. The algorithm is based on a variant of the Kalman Filter named the square root and extended version Information Filter and is oriented to be used in real time applications.

Keywords: - Parallel algorithms, RLS, Kalman, Information Filter, OpenMP, MPI, Shared Distributed Memory Multiprocessors

1. Introduction

In some signal processing applications like adaptive filtering it is necessary to solve the Recursive Least Squares problem. Some examples are channel or multichannel equalization, echo cancellation, ...

Parallel algorithms for solving the RLS problem using mainly systolic architectures can be found in [5], [6], [7], ... In this paper we propose a parallel algorithm for solving this problem, based on a variant of the Kalman filter, [1], using shared and/or distributed memory parallel architectures.

The proposed parallel algorithm can be used in heterogenous systems without loss of parallel algorithmic properties.

First, we will show a variant of the Kalman filter to solve the Recursive Least Squares problem. Then the sequential algorithm will be stated and finally, the parallel algorithm will be derived showing some experimental results.

Notation: Boldface typing denotes a vector or a matrix, otherwise the variable will be scalar. A superscript with an asterisc (*) denotes the conjugate transposed of a matrix.

1.1. Relation between the Kalman filter and the Recursive Least Squares problem

The Kalman filter can be used to solve the recursive state-space estimation of a system. We consider the next state-space model that is useful to relate the recursive state-space estimation problem with the recursive least squares problem [1]:

$$\begin{aligned}\mathbf{x}_{i+1} &= \lambda^{-1/2}\mathbf{x}_i \\ \mathbf{y}_i &= \mathbf{H}_i\mathbf{x}_i + \mathbf{v}_i\end{aligned}$$

where $\mathbf{x}_i \in \mathbb{C}^{n\times 1}$ is the state vector, $\mathbf{y}_i \in \mathbb{C}^{q\times 1}$ is the observation vector, $\mathbf{v}_i \in \mathbb{C}^{q\times 1}$ is the observation noise vector, $\mathbf{H}_i \in \mathbb{C}^{q\times n}$ is the measurement matrix, $0 \ll \lambda \leq 1$, $q \ll n$, and $i \geq 0$. The Kalman filter computes recursively the linear least-mean squares estimation of the state with the knowledge of the observations and some system parameters.

There are many algorithmic variants of the Kalman filter [1], [2]: the information filter, the square root Kalman filter or covariance filter, the extended square-root information filter, the square-root Chandrasekhar filter, the explicit Chandrasekhar filter, ... We are interested in the extended square-root information filter due to its simplicity and its high parallelization potential. The details of this algorithm are:

for $i = 0, 1, 2, \ldots$

$$\begin{aligned} \mathbf{A}_i\mathbf{\Theta}_i &= \mathbf{B}_i & (1)\\ \hat{\mathbf{x}}_{i+1} &= \lambda^{-1/2}\hat{\mathbf{x}}_i + \overline{\mathbf{K}}_{p,i}\left[\mathbf{R}_{e,i}^{-1/2}\mathbf{e}_i\right] & (2) \end{aligned}$$

end for

where

$$\mathbf{A}_i = \begin{pmatrix} \lambda^{1/2}\mathbf{P}_i^{-*/2} & \lambda^{1/2}\mathbf{H}_i^* \\ \hat{\mathbf{x}}_i^*\mathbf{P}_i^{-*/2} & \mathbf{y}_i^* \\ \lambda^{-1/2}\mathbf{P}_i^{1/2} & \mathbf{0} \end{pmatrix}$$

$$\mathbf{B}_i = \begin{pmatrix} \mathbf{P}_{i+1}^{-*/2} & \mathbf{0} \\ \hat{\mathbf{x}}_{i+1}^*\mathbf{P}_{i+1}^{-*/2} & \mathbf{e}_i^*\mathbf{R}_{e,i}^{-*/2} \\ \mathbf{P}_{i+1}^{1/2} & -\overline{\mathbf{K}}_{p,i} \end{pmatrix}$$

and $\hat{\mathbf{x}}_i \in \mathbb{C}^{n\times 1}$, $\hat{\mathbf{x}}_0 = \overline{\mathbf{x}}_0$, $\mathbf{P}_i \in \mathbb{C}^{n\times n}$, $\mathbf{P}_0^{-*/2} = \mathbf{\Pi}_0^{-*/2}$, $\mathbf{H}_i \in \mathbb{C}^{q\times n}$, $\mathbf{y}_i \in \mathbb{C}^{q\times 1}$, $\mathbf{e}_i = \mathbf{y}_i - \mathbf{H}_i\hat{\mathbf{x}}_i \in \mathbb{C}^{q\times 1}$, $\mathbf{R}_{e,i} = \mathrm{cov}(\mathbf{e}_i) \in \mathbb{C}^{q\times q}$, and $\mathbf{K}_{p,i} = \mathrm{cov}(\mathbf{x}_{i+1}, \mathbf{e}_i) \in \mathbb{C}^{n\times q}$. Usually $q \ll n$. $\mathbf{\Theta}_i$ is an orthogonal transformation that partially lower triangularizes the matrix that postmultiplies and can be got with a sequence of Givens rotations. If $\mathbf{P}_i^{-1}$ is positive definite, then we can factorize it as the product of its Cholesky triangles $\mathbf{P}_i^{-1} = \mathbf{P}_i^{-*/2}\mathbf{P}_i^{-1/2}$, so $\mathbf{P}_i^{1/2}$ is upper triangular and $\mathbf{P}_i^{-*/2}$ is lower triangular.

In [1], [2] we can find that the least squares problem $\mathbf{x}_{LS}$ related to $\mathbf{y} = \mathbf{K}\mathbf{x}_0 + \mathbf{v}$ can be solved recursively with a Kalman filter if we rewrite $\mathbf{y} = \mathbf{K}\mathbf{x}_0 + \mathbf{v} = \mathbf{\Delta}\mathbf{H}\mathbf{x}_0 + \mathbf{v}$, with $\mathbf{\Delta} = \mathrm{diag}(1, \lambda^{-1/2}, \lambda^{(-1/2)^2}, \ldots)$, and considering $\hat{\mathbf{x}}_i = (\lambda^{-1/2})^i\hat{\mathbf{x}}_0$, $\mathbf{P}_0^{-1} = \epsilon\mathbf{I}_n$, with ϵ a sufficient small positive number, and $\mathbf{x}_{LS} = \hat{\mathbf{x}}_0$.

We can rewrite the algorithm in a more convenient way:

$$\mathbf{C}_0 = \begin{pmatrix} \mathbf{P}_0^{-*/2} \\ \hat{\mathbf{x}}_0^*\mathbf{P}_0^{-*/2} \\ \mathbf{P}_0^{1/2} \end{pmatrix}; \quad \Lambda = \begin{pmatrix} \lambda^{1/2}\mathbf{I}_n & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & 1 & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \lambda^{-1/2}\mathbf{I}_n \end{pmatrix}$$

for $i = 0, 1, \ldots$

$$\left(\Lambda\mathbf{C}_i, \begin{pmatrix} \lambda^{1/2}\mathbf{H}_i^* \\ \mathbf{y}_i^* \\ \mathbf{0} \end{pmatrix}\right)\mathbf{\Theta}_i = \left(\mathbf{C}_{i+1}, \begin{pmatrix} \mathbf{0} \\ \mathbf{e}_i^*\mathbf{R}_{e,i}^{-*/2} \\ -\overline{\mathbf{K}}_{p,i} \end{pmatrix}\right)$$

$$\hat{\mathbf{x}}_{i+1} = \lambda^{-1/2}\hat{\mathbf{x}}_i + \overline{\mathbf{K}}_{p,i}\left[\mathbf{R}_{e,i}^{-1/2}\mathbf{e}_i\right]$$

$$\hat{\mathbf{x}}_0 = (\lambda^{1/2})^{i+1}\hat{\mathbf{x}}_{i+1}$$

end for

The cost of the iteration is $O(qn^2)$, when $q \ll n$, [1], [2], [4].

2. Parallel algorithm

Let us suppose that we have $p = 3$ processors: P_0, P_1, and P_2. The parallel algorithm must compute the equations (1) and (2).

2.1. Data partition

A matrix or vector enclosed within square brackets with a processor subscript denotes that part of the matrix or vector is in such processor. If it is enclosed within parenthesis then it denotes that the entire matrix or vector is in such processor.

The vector $\hat{\mathbf{x}}_i$ will be (always) in the last processor (P_2 in this case). The $\mathbf{C}_i$ matrix will be divided by columns (n_0 columns belong to P_0, n_1 columns to P_1, and n_2 columns to P_2, with $n_0 + n_1 + n_2 = n$). The last q columns of $\mathbf{A}_i$ denoted as $\mathbf{D}_i$ will be initially in P_0. So the initial data partition will be as shown:

$$\mathbf{A}_i = \left(\Lambda\,[\mathbf{C}_i]_{P_0} \quad \Lambda\,[\mathbf{C}_i]_{P_1} \quad \Lambda\,[\mathbf{C}_i]_{P_2} \quad (\mathbf{D}_i)_{P_0} \right)$$

$\mathbf{D}_i$ will be manipulated in a pipelined way by all the processors, so the processor subscript will change accordingly. The data prepared by P_j for P_{j+1} will be denoted as $(\mathbf{D}_i^+)_{P_j}$, and these data in the P_{j+1} memory space will be denoted as $(\mathbf{D}_i^-)_{P_{j+1}}$. Again, the data prepared by P_{j+1} for P_{j+2} based on $(\mathbf{D}_i^-)_{P_{j+1}}$ will be denoted as $(\mathbf{D}_i^+)_{P_{j+1}}$ and so on.

Inside $(\mathbf{D}_i)_{P_0}^-$, $\lambda^{1/2}\mathbf{H}_i^*$ will be divided by rows in p parts (three in this example with n_0, n_1, and n_2

rows respectively). The zero submatrix of $(\mathbf{D}_i)^-_{P_0}$ will be divided in the same parts. We will observe that the number of the nonzero elements of $\left(\mathbf{D}_i^{-/+}\right)_{P_j}$ will be always $q(n+1)$ in any P_j at any time.

2.2. Processors tasks

Let us suppose that P_0 gets zeroes in the first n_0 rows of $\lambda^{1/2}\mathbf{H}_i^*$ by applying a sequence of Givens rotations denoted by $\mathbf{\Theta}_{i,P_0}$:

$$\begin{aligned}
\mathbf{A}_i' &= \mathbf{A}_i\mathbf{\Theta}_{i,P_0}\\
&= \begin{pmatrix} \Lambda\,[\mathbf{C}_i]_{P_0} & \Lambda\,[\mathbf{C}_i]_{P_1} & \Lambda\,[\mathbf{C}_i]_{P_2} & \left(\mathbf{D}_i^-\right)_{P_0} \end{pmatrix}\Theta_{i,P_0}\\
&= \begin{pmatrix} \Lambda\,[\mathbf{C}_i]_{P_0} & \Lambda\,[\mathbf{C}_i]_{P_1} & \Lambda\,[\mathbf{C}_i]_{P_2} & \begin{pmatrix} \left(\lambda^{1/2}\mathbf{H}_i^*\right)_{n_0}\\ \left(\lambda^{1/2}\mathbf{H}_i^*\right)_{n_1}\\ \left(\lambda^{1/2}\mathbf{H}_i^*\right)_{n_2}\\ \mathbf{y}_i^*\\ (\mathbf{0})_{n_0}\\ (\mathbf{0})_{n_1}\\ (\mathbf{0})_{n_2} \end{pmatrix}_{P_0} \end{pmatrix}\Theta_{i,P_0}\\
&= \begin{pmatrix} [\mathbf{C}_{i+1}]_{P_0} & \Lambda\,[\mathbf{C}_i]_{P_1} & \Lambda\,[\mathbf{C}_i]_{P_2} & \begin{pmatrix} (\mathbf{0})_{n_0}\\ \left(\lambda^{1/2}\mathbf{H}_i^*\right)'_{n_1}\\ \left(\lambda^{1/2}\mathbf{H}_i^*\right)'_{n_2}\\ \mathbf{y}_i^{*'}\\ (\mathbf{L}_i)_{n_0}\\ (\mathbf{0})_{n_1}\\ (\mathbf{0})_{n_2} \end{pmatrix}_{P_0} \end{pmatrix}\\
&= \begin{pmatrix} [\mathbf{C}_{i+1}]_{P_0} & \Lambda\,[\mathbf{C}_i]_{P_1} & \Lambda\,[\mathbf{C}_i]_{P_2} & \left(\mathbf{D}_i^+\right)_{P_0} \end{pmatrix}
\end{aligned}$$

We can observe that the data not belonging to P_0 are not involved in the operations. The n_0 rows of $\left(\lambda^{1/2}\mathbf{H}_i^*\right)_{n_0}$ are zeroed and n_0 rows of nonzero elements, $(\mathbf{L}_i)_{n_0}$, appear below $\mathbf{y}_i^{*'}$, as a consequence of the Givens rotation sequence application. We can observe too that the first n_0 columns of the result $\mathbf{A}_i'$ are the first n_0 columns of the matrix $\mathbf{A}_{i+1}$ (except the $\lambda^{1/2}$ and $\lambda^{-1/2}$ factors). This is useful to get a pipelined behaviour in the work of the processors.

Now, if P_0 transfers $\left(\lambda^{1/2}\mathbf{H}_i^*\right)'_{n_1}$, $\left(\lambda^{1/2}\mathbf{H}_i^*\right)'_{n_2}$, $\mathbf{y}_i^{*'}$, and $(\mathbf{L}_i)_{n_0}$, —these are the $q(n+1)$ floating point numbers that form the nonzero part of $(\mathbf{D}_i)^+_{P_0}$— to P_1, naming it as $(\mathbf{D}_i)^-_{P_1}$, then

$$\begin{aligned}
\mathbf{A}_i'' &= \mathbf{A}_i'\mathbf{\Theta}_{i,P_1}\\
&= \begin{pmatrix} [\mathbf{C}_{i+1}]_{P_0} & \Lambda\,[\mathbf{C}_i]_{P_1} & \Lambda\,[\mathbf{C}_i]_{P_2} & \left(\mathbf{D}_i^-\right)_{P_1} \end{pmatrix}\mathbf{\Theta}_{i,P_1}\\
&= \begin{pmatrix} [\mathbf{C}_{i+1}]_{P_0} & \Lambda\,[\mathbf{C}_i]_{P_1} & \Lambda\,[\mathbf{C}_i]_{P_2} & \begin{pmatrix} (\mathbf{0})_{n_0}\\ \left(\lambda^{1/2}\mathbf{H}_i^*\right)'_{n_1}\\ \left(\lambda^{1/2}\mathbf{H}_i^*\right)'_{n_2}\\ \mathbf{y}_i^{*'}\\ (\mathbf{L}_i)_{n_0}\\ (\mathbf{0})_{n_1}\\ (\mathbf{0})_{n_2} \end{pmatrix}_{P_1} \end{pmatrix}\mathbf{\Theta}_{i,P_1}\\
&= \begin{pmatrix} [\mathbf{C}_{i+1}]_{P_0} & [\mathbf{C}_{i+1}]_{P_1} & \Lambda\,[\mathbf{C}_i]_{P_2} & \begin{pmatrix} (\mathbf{0})_{n_0}\\ (\mathbf{0})_{n_1}\\ \left(\lambda^{1/2}\mathbf{H}_i^*\right)''_{n_2}\\ \mathbf{y}_i^{*''}\\ (\mathbf{L}_i)'_{n_0}\\ (\mathbf{M}_i)_{n_1}\\ (\mathbf{0})_{n_2} \end{pmatrix}_{P_1} \end{pmatrix}\\
&= \begin{pmatrix} [\mathbf{C}_{i+1}]_{P_0} & [\mathbf{C}_{i+1}]_{P_1} & \Lambda\,[\mathbf{C}_i]_{P_2} & \left(\mathbf{D}_i^+\right)_{P_1} \end{pmatrix}
\end{aligned}$$

P_1 can get already zeroes in the n_1 rows of $\left(\lambda^{1/2}\mathbf{H}_i^*\right)'_{n_1}$ with the application of the Givens rotation sequence $\mathbf{\Theta}_{i,P_1}$ and P_0 could work already with its part of $\mathbf{A}_{i+1}$ (pipelined algorithm). Again, if P_1 transfers the nonzero part of $(\mathbf{D}_i)^+_{P_1}$, to P_2 — $q(n+1)$ floating point numbers—, then P_2 can get already zeroes in the n_2 rows of $\left(\lambda^{1/2}\mathbf{H}_i^*\right)''_{n_2}$ with the application of the Givens rotation sequence $\mathbf{\Theta}_{i,P_2}$ and P_1 could work already with its part of $\mathbf{A}_{i+1}$ (pipelined algorithm):

$$\begin{aligned}
\mathbf{A}_i''' &= \mathbf{B}_i\\
&= \mathbf{A}_i''\mathbf{\Theta}_{i,P_2}\\
&= \begin{pmatrix} [\mathbf{C}_{i+1}]_{P_0} & [\mathbf{C}_{i+1}]_{P_1} & \Lambda\,[\mathbf{C}_i]_{P_2} & \left(\mathbf{D}_i^-\right)_{P_2} \end{pmatrix}\mathbf{\Theta}_{i,P_2}\\
&= \begin{pmatrix} [\mathbf{C}_{i+1}]_{P_0} & [\mathbf{C}_{i+1}]_{P_1} & \Lambda\,[\mathbf{C}_i]_{P_2} & \begin{pmatrix} (\mathbf{0})_{n_0}\\ (\mathbf{0})_{n_1}\\ \left(\lambda^{1/2}\mathbf{H}_i^*\right)''_{n_2}\\ \mathbf{y}_i^{*''}\\ (\mathbf{L}_i)'_{n_0}\\ (\mathbf{M}_i)_{n_1}\\ (\mathbf{0})_{n_2} \end{pmatrix}_{P_2} \end{pmatrix}\mathbf{\Theta}_{i,P_2}\\
&= \begin{pmatrix} [\mathbf{C}_{i+1}]_{P_0} & [\mathbf{C}_{i+1}]_{P_1} & [\mathbf{C}_{i+1}]_{P_2} & \begin{pmatrix} (\mathbf{0})_{n_0}\\ (\mathbf{0})_{n_1}\\ (\mathbf{0})_{n_2}\\ \mathbf{y}_i^{*'''}\\ (\mathbf{L}_i)''_{n_0}\\ (\mathbf{M}_i)'_{n_1}\\ (\mathbf{N})_{n_2} \end{pmatrix}_{P_2} \end{pmatrix}\\
&= \begin{pmatrix} [\mathbf{C}_{i+1}]_{P_0} & [\mathbf{C}_{i+1}]_{P_1} & [\mathbf{C}_{i+1}]_{P_2} & \begin{pmatrix} (\mathbf{0})_{n_0+n_1+n_2=n}\\ \mathbf{e}_i^*\mathbf{R}_{e,i}^{-*/2}\\ -\overline{\mathbf{K}}_{p,i} \end{pmatrix}_{P_2} \end{pmatrix}
\end{aligned}$$

At the end, we got the expression in (1) by means

$$\mathbf{A}_i\mathbf{\Theta}_i = \mathbf{A}_i\mathbf{\Theta}_{i,P_0}\mathbf{\Theta}_{i,P_1}\mathbf{\Theta}_{i,P_2} = \mathbf{B}_i$$

in a pipelined way. At this time, P_2 can update de solution $\hat{\mathbf{x}}_0$ (or $\mathbf{x}_{LS}$):

$$\begin{aligned}\hat{\mathbf{x}}_{i+1} &= \lambda^{-1/2}\hat{\mathbf{x}}_i + \overline{\mathbf{K}}_{p,i}\left[\mathbf{R}_{e,i}^{-1/2}\mathbf{e}_i\right]\\ \hat{\mathbf{x}}_0 &= (\lambda^{1/2})^{i+1}\hat{\mathbf{x}}_{i+1}\end{aligned}$$

Figure 1 shows graphically the pipelined parallel algorithm. τ represents the data saving from one iteration to the next, and initial values.

It is easy to extrapolate the tasks to a different number of processors in the pipeline: the first gets $\mathbf{H}_i$ and $\mathbf{y}_i$ from the acquisition subsystem, gets some zeroes in the last q columns of $\mathbf{A}_i$, and transfers these columns to the second processor. This gets some zeroes in these columns and transfers them to the next one, ..., and the last one, gets the last zeroes and updates the solution. We can see that there are three kind of processors or tasks: the *first*, the *intermediate* and the *last*. If there are only two processors, only the *first* and the *final* processor would exist.

2.3. Arithmetic cost

Le us suppose that n_j columns of $\mathbf{A}_i$ (or $\mathbf{C}_i$) have been assigned to the P_j processor.

1. If P_j is the *first* processor then it must get the data $\mathbf{H}_i$ and $\mathbf{y}_i$, and multiply $\mathbf{H}_i^*$ by $\lambda^{1/2}$: qn floating point multiplications, so the cost is $O(qn)$.
2. P_j must multiply $\left(\mathbf{P}_i^{-*/2}\right)_{P_j}$ by $\lambda^{1/2}$, and $\left(\mathbf{P}_i^{1/2}\right)_{P_j}$ by $\lambda^{-1/2}$: $n_j(n+1)$ floating point multiplications, so $O(n_jn)$.
3. P_j must get qn_j zeroes in the last q columns of $\mathbf{A}_i$, calculating qn_j Givens rotations and aplying each of them to $n+2$ entries. So the cost is $O(qn_jn)$.
4. If P_j is the *last* processor then it must update the solution $\hat{\mathbf{x}}_0$, so the cost is $O(q^2n)$.

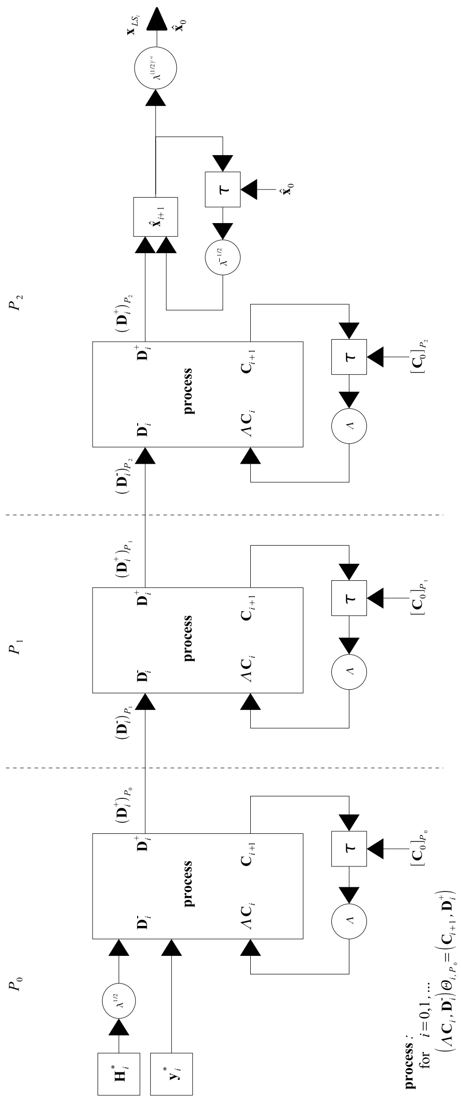

Figure 1: Pipelined parallel algorithm for the RLS problem.

So it does not matter if P_j is either the first or an intermediate or the last processor: the cost is $O(qn_jn)$ due to steps 2. and 3. mainly.

2.4. Load balancing

In the pipelined algorithm, the processors load must be *tuned* to avoid a processor is waiting, so the best a priori load balancing criterion is $n_j = n/p$, where p is the number of processors, so the cost would be $O(qn^2/p)$. This criterion can be refined to get a more perfect load balancing. If we compare this result with the results of the subsection 1.1 we can get a maximum relative speedup of

p. So if we get a perfect load balancing, the definitive efficiency or speedup will depend on the interprocessor communication overload as explained in the next subsection.

2.4.1. Automatic/adaptive load balancing and heterogeneous systems

We can reconfigure the load of every processor assigning more or less columns of $\mathbf{C}_i$ to P_j ($[\mathbf{C}_i]_{P_j}$). We can do it before the algorithm is running (in a test process —automatic load balancing—) or when the algorithm is running —adaptive or dynamic load balancing—. In this case it is necessary to transfer the columns that will not be used by a processor to the apropriate neighbour.

This scheme of variable load per processor is suitable for using it with a heterogenous system: the number of columns of $\mathbf{C}_i$ assigned to a processor should be proportional to its computational power with the condition that all the processors take the same time in its execution, so an added (perhaps slower) processor will involve a new distribution of less columns per processor (less execution time).

2.5. Communication costs and multiprocessor systems

A processor P_j must *transfer* the nonzero part of $\left(\mathbf{D}_i^+\right)_{P_j}$ to P_{j+1}. These are $q(n+1)$ floating point numbers.

If we use a distributed memory multiprocessor message passing system using for example MPI, then the cost is the time used to do this transfer that depends on the network topology and the size of this submatrix. Every processor of the figure 1 can be mapped in a processor of this system.

If we use a shared memory multiprocessor system then every processor of the figure 1 can be mapped in a processor of a shared memory multiprocessor using for instance OpenMP. In this case, the data transfer can consist in the copy of this data from the memory space of P_j to the memory space of P_{j+1} and control the access to it with a typical producer-consumer strategy. The copy of the data to the next processor memory space can be an important time consuming operation. We avoid this copy time using a circular buffer whose elements are arrays of size $q(n+1)$ floating point numbers, so the symbolic copy can consist in a pointer to an element buffer updating. For example, if $\left(\mathbf{D}_i^-\right)_{P_j}$ denotes the data that is being prepared by P_j for P_{j+1} in the i iteration and $\left(\mathbf{D}_i^+\right)_{P_j}$ denotes de data already prepared by P_j for P_{j+1}, then figures 2 and 3 show graphically the process of the *symbolic* copy for P_j when it has finished its work for the next processor and the data prepared by the preceeding one is available.

We extrapolate this ideas to a shared and distributed memory multiprocessors with minimum change in the code.

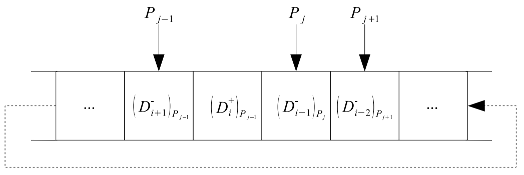
P_{j-1}
P_j
P_{j+1}
...
$\left(D^-_{i+1}\right)_{P_{j-1}}$
$\left(D^+_i\right)_{P_{j-1}}$
$\left(D^-_{i-1}\right)_{P_j}$
$\left(D^-_{i-2}\right)_{P_{j+1}}$
...

Figure 2: Data is already available for P_j

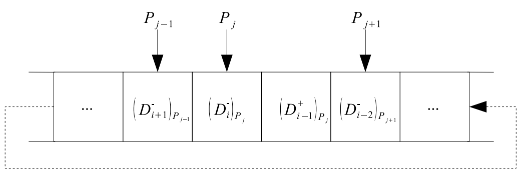
P_{j-1}
P_j
P_{j+1}
...
$\left(D^-_{i+1}\right)_{P_{j-1}}$
$\left(D^-_i\right)_{P_j}$
$\left(D^+_{i-1}\right)_{P_j}$
$\left(D^-_{i-2}\right)_{P_{j+1}}$
...

Figure 3: Data is already available for P_{j+1} $(i-1)$-iteration . P_j is working for P_{j+1} i-iteration.

3. Experimental results

The hardware was a cluster of two SMP made up by two Intel(R) Xeon(TM) 2.20 GHz and 512 kB cache CPUs and 4GB of total memory. The operating system was Red Hat 9 with a minimum number of services running. We parallelized the algorithm using MPI (MPICH 1.2.6) and OpenMP with the Intel Fortran Compiler 8.0 using code optimization and openmp options. The communication among the threads of a SMP was by means a buffer written and readed by them, using some variables to control the access. The communication among SMP nodes was by means of the send a receive MPI functions. The load balancing criterion was to assign the same number of columns to each processor. Other algorithm parameters were $\lambda = 0.99999$ and $q = 1$.

3.1. Scaled relative speedup

The parallel and sequential times considered are the algorithm time per iteration. This time is calculated obtaining the mode of all the iteration times in a simulation in order to minimize the measurements variance. Figure 4 shows the scaled relative speedup S in the Y-axis of the parallel system in four situations: ideal, using OpenMP in a 2 processor SMP, using MPI in a cluster of two nodes (with 1 processor per node) and using MPI+OpenMP in a cluster of 2 nodes (with 2 processor per node). In the X-axis appears de total number of processors of the parallel machine and the size n of the problem in order to get a scaled speedup. We can observe how speedup increases such that the efficiency is approximately maintained in the considered range of test.

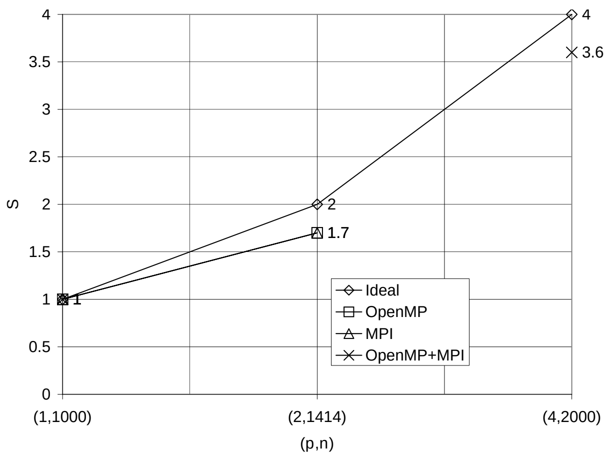

Figure 4: Scaled relative speedup

4. Conclusions

We propose a parallel algorithm for solving the RLS problem with a good scalability in the considered range This parallel algorithm has been implemented without important changes in three multiprocessor architectures: distributed, shared and mixed memory systems. We got a very good performance in a relatively cheap system like a biprocessor SMP cluster.

Acknowledgments: This work has been supported by spanish MCYT and FEDER under Grant TIC2003-08238-C02

References:

[1] Ali H. Sayed and Thomas Kailath, A State-Space Approach to Adaptive RLS Filtering *IEEE Signal Processing Magazine*, pp. 18–60, July 1994

[2] Thomas Kailath, Ali H. Sayed and Babak Hassibi, *Linear Estimation*, Prentice Hall Information and System Sciences Series, 2000

[3] Djigan, V.I. Fast RLS with Parallel Computations. *IEEE 7th CAS Symposium on Emerging Technologies: Circuits and Systems for 4G Mobile Wireless Communications*, pp. 42–45, June 2005.

[4] F.J. Martínez Zaldívar, A.M. Vidal Maciá and A. González Salvador. A parallel algorithm based on a variant of the Kalman Filter for solving the RLS problem. *WSEAS Transactions on Circuits and Systems*, Vol. 3, pp. 2143–2148, Dec. 2004

[5] M. Moonen and E. Deprettere, A fully pipelined RLS-based array for channel equalization, *Journal of VLSI Signal Processing*, Vol. 14, No. 1, October 1996, pp 67-74.

[6] M. Moonen Systolic algorithms for recursive total least squares estimation and mixed RLS/RTLS, *International Journal of High Speed Electronics, special issue on 'Massively Parallel Computing'*, Vol. 4 (1993), No. 1, pp 55-68.

[7] Lu, M. Qiao, X. Chen, G., A parallel square-root algorithm for modified extended Kalman filter. *IEEE Transactions on Aerospace and Electronic Systems*, Vol. 28, Issue 1, pp. 153–163, 1992