Efficient Parallel Algorithm for Constructing a Unit Triangular Matrix with Prescribed Singular Values

Georgina Flores-Becerra[12], Victor M. Garcia[1], and Antonio M. Vidal[1]

[1] Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Camino de Vera s/n, 46022 Valencia, España
{gflores, vmgarcia, avidal}@dsic.upv.es
[2] Departamento de Sistemas y Computación. Instituto Tecnológico de Puebla
Av. Tecnológico 420, Col. Maravillas, C.P. 72220, Puebla, México

Abstract. The problem tackled in this paper is the parallel construction of a unit triangular matrix with prescribed singular values, when these fulfill Weyl's conditions [9]; this is a particular case of the Inverse Singular Value Problem. A sequential algorithm for this problem was proposed in [10] by Kosowsky and Smoktunowicz. In this paper parallel versions of this algorithm will be described, both for shared memory and distributed memory architectures. The proposed parallel implementation is better suited for the shared memory paradigm; this is confirmed by the numerical experiments; the shared memory version, reaches an efficiency over 90%, and reduces substantially the execution times compared with the sequential algorithm.

1 Introduction

Inverse problems can be found in many branches of Science and Engineering, such as simulation of mechanical systems, geophysics, tomography, and many others [3, 6, 11, 12]. A particular instance of this family of problems is the Inverse Singular Value Problem (ISVP), which can be defined as:

Given a set of n positive real numbers $S^* = \{s_1^*, s_2^*, ..., s_n^*\}$, where $s_1^* \geq s_2^* \geq ... \geq s_n^*$, find a matrix $A \in \Re^{nxn}$, with a certain structure, whose singular values are S^*.

There exist several algorithms to solve this problem, such as MI, MIII, EP and FB [5], which are iterative Newton-like algorithms, with high computational cost ($O(n^4)$ for MI, MIII and EP; and $O(n^6)$ for FB). If the desired matrix must have a certain structure, the computational costs can be drastically reduced. As an example, the ISVP problem for Toeplitz matrices can be solved with Newton algorithms with cost $O(n^3)$.

The problem of the construction of a unit lower triangular matrix $A \in R^{n\times n}$, such that the singular values of A are $s_1^* \geq s_2^* \geq ... \geq s_n^*$, was proposed by Kosowski and Smoktunowicz in [10]. It can be seen as a special case of the

ISVP which could be named Inverse Unit Triangular Singular Value Problem (IUTSVP). The existence of solution was given by Horn [9], who proved that such matrix A exists if and only if the following conditions are fulfilled (these are called Weyl conditions):

$$s_1^* s_2^* s_3^* ... s_i^* \geq 1, \quad (i = 2 : n) \quad and \quad s_1^* s_2^* s_3^* ... s_n^* = 1.$$

Kosowski and Smoktunowicz proposed an $O(n^2)$ algorithm (based on Horn's proof) to solve the IUTSVP. It is a direct algorithm (that is, it solves the problem in a finite number of steps), in contrast with the iterative methods needed to solve the general ISVP.

This paper is focused on the design of a parallel version of the algorithm proposed by Kosowski and Smoktunowicz, and its implementation for shared memory and distributed memory computers; of course, the primary goal is the reduction of the time needed to solve this problem. Both implementations are compared from different points of view.

This paper is organized as follows: The theoretical background, along with the sequential algorithm are shown in the Section 2. In Section 3 the distributed memory parallel algorithm is introduced and discussed, and the shared memory algorithm is discussed in Section 4. In these three sections numerical results are given. Finally, in Section 5 the results obtained are compared and analyzed, offering the conclussions of the study.

2 Method based in Weyl's conditions(WE Method)

The method proposed by Kosowski and Smoktunowicz to solve IUTSVP is based on the construction of a sequence of unit lower triangular matrices $A^{(i)}$ $(i = 1 : n)$ equivalent to the diagonal matrix $diag(s_1^*, s_2^*, ..., s_n^*)$[3]. To build the matrices of this sequence the following lemma is applied:

Lemma 1. *Two real numbers $s_i^*, s_j^* > 0$ such that $s_i^* \geq 1 \geq s_j^*$ or $s_j^* \geq 1 \geq s_i^*$, are the singular values of the matrix*

$$\begin{bmatrix} 1 & 0 \\ \sqrt{(s_i^{*2} - 1)(1 - s_j^{*2})} & s_i^* s_j^* \end{bmatrix}.$$

This lemma leads to take submatrices 2×2 of $A^{(i)}$ $(i = 1 : n)$ in the form $diag(d_i, d_j)$ such that d_i, d_j fulfill

$$d_i \geq 1 \geq d_j \quad or \quad d_j \geq 1 \geq d_i. \tag{1}$$

To ensure that (1) is fulfilled, Kosowski et.al. apply the next lemma:

[3] Two matrices M and N are unitarily equivalent if exist unitary matrices U, V such that $M = UNV^t$; under these conditions M and N shall have the same singular values.

Lemma 2. *If the real numbers $s_1^* \geq s_2^* \geq ... \geq s_n^* > 0$ satisfy Weyl's conditions, then there exists a permutation $\{d_1, d_2, ..., d_n\}$ of $\{s_1^*, s_2^*, ..., s_n^*\}$ such that*

$$d_1 d_2 ... d_{i-1} \geq 1 \geq d_i \quad or \quad d_i \geq 1 \geq d_1 d_2 ... d_{i-1} \quad (i = 2 : n). \tag{2}$$

Given the matrix $A^{(1)} = diag(d_1, d_2, ..., d_n)$, and if d_1 and d_2 satisfy (1), then the following matrix exists:

$$L^{(2)} = \begin{bmatrix} 1 & 0 \\ \sqrt{(d_1^2 - 1)(1 - d_2^2)} & d_1 d_2 \end{bmatrix}$$

with singular values d_1, d_2. Then, we can build $A^{(2)} = diag(L^{(2)}, D^{(1)}_{n-2\times n-2})$, where $D^{(1)}_{n-2\times n-2} = diag(d_3, d_4, ..., d_n)$. $A^{(2)}$ is equivalent to $A^{(1)}$ because there exist 2×2 unitary matrices $U^{(2)}$, $V^{(2)}$ such that $L^{(2)} = U^{(2)} diag(d_1, d_2) V^{(2)T}$.

Once $A^{(2)}$ has been built, starts an iterative process to build $A^{(3)}, A^{(4)}, ...A^{(n)}$. For example, the construction of $A^{(3)}$ is based on the singular value decomposition (SVD) of the 2×2 matrix $L^{(3)}$:

$$L^{(3)} = \begin{bmatrix} 1 & 0 \\ \sqrt{(d_1^2 d_2^2 - 1)(1 - d_3^2)} & d_1 d_2 d_3 \end{bmatrix} = U^{(3)} \; diag(d_1 d_2, \; d_3) \; V^{(3)T} \tag{3}$$

and the $A^{(3)}$ can be written as $A^{(3)} = Q^{(3)} A^{(2)} Z^{(3)T}$, where:

$$Q^{(3)} = diag(I_{1\times 1}, U^{(3)}, I_{n-3\times n-3}), \quad Z^{(3)T} = diag(I_{1\times 1}, V^{(3)T}, I_{n-3\times n-3}), \tag{4}$$

$$A^{(2)} = \left[\begin{array}{c|cc|ccc} 1 & 0 & 0 & 0 & ... & 0 \\ \hline \sqrt{(d_1^2 - 1)(1 - d_2^2)} & d_1 d_2 & 0 & 0 & ... & 0 \\ 0 & 0 & d_3 & 0 & ... & 0 \\ \hline 0 & 0 & 0 & d_4 & ... & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & ... & d_n \end{array}\right] = \left[\begin{array}{c|c|c} B^{(2)}_{1\times 1} & & \\ \hline C^{(2)}_{2\times 1} & diag(d_1 d_2, d_3) & \\ \hline & & D^{(2)}_{n-3\times n-3} \end{array}\right] \tag{5}$$

(See eq. (8) for the definition of B) and, performing the matrix multiplications, $A^{(3)}$ can be written as:

$$A^{(3)} = \left[\begin{array}{c|c|c} B^{(2)}_{1\times 1} & & \\ \hline U^{(3)} C^{(2)}_{2\times 1} & L^{(3)} & \\ \hline & & D^{(2)}_{n-3\times n-3} \end{array}\right]. \tag{6}$$

The same procedure is followed to compute $A^{(4)}, A^{(5)}, ..., A^{(n)}$. The final result will be the unit lower triangular matrix $A^{(n)}$, whose singular values are S^*. Therefore, if the numbers p_i, z_i are defined as follows:

$$p_i = d_1 d_2 ... d_i, \quad (i = 1 : n); \quad and \quad z_i = \sqrt{(p_{i-1}^2 - 1)(1 - d_i^2)}, \quad (i = 2 : n); \tag{7}$$

$A^{(n)}$ has the form:

$$A^{(n)} = \left[\begin{array}{c|c} B^{(n-1)}_{n-2\times n-2} & \\ \hline U^{(n)}_{2\times 2}C^{(n-1)}_{2\times n-2} & L^{(n)}_{2\times 2} \end{array}\right],$$

where $B^{(n-1)}_{n-2\times n-2}$, $U^{(n)}_{2\times 2}C^{(n-1)}_{2\times n-2}$ and $L^{(n)}_{2\times 2}$ are:

$$B^{(n-1)}_{n-2\times n-2} = \left[\begin{array}{c|c|c|c} 1 & 0 & ... & 0 \\ \hline u^{(3)}_{11}z_2 & 1 & ... & 0 \\ \hline u^{(4)}_{11}u^{(3)}_{21}z_2 & u^{(4)}_{11}z_3 & ... & 0 \\ \hline \vdots & \vdots & \ddots & \vdots \\ \hline u^{(n-1)}_{11}u^{(n-2)}_{21}...u^{(3)}_{21}z_2 & u^{(n-1)}_{11}u^{(n-2)}_{21}...u^{(4)}_{21}z_3 & ... & 1 \end{array}\right] \tag{8}$$

$$U^{(n)}_{2\times 2}C^{(n-1)}_{2\times n-2} = \left[\begin{array}{c|c|c|c} u^{(n)}_{11}u^{(n-1)}_{21}...u^{(3)}_{21}z_2 & u^{(n)}_{11}u^{(n-1)}_{21}...u^{(4)}_{21}z_3 & ... & u^{(n)}_{11}z_{n-1} \\ \hline u^{(n)}_{21}u^{(n-1)}_{21}...u^{(3)}_{21}z_2 & u^{(n)}_{21}u^{(n-1)}_{21}...u^{(4)}_{21}z_3 & ... & u^{(n)}_{21}z_{n-1} \end{array}\right] \tag{9}$$

$$L^{(n)}_{2\times 2} = \left[\begin{array}{cc} 1 & 0 \\ z_n & p_n \end{array}\right] = \left[\begin{array}{cc} 1 & 0 \\ z_n & 1 \end{array}\right]. \tag{10}$$

From these equations (8), (9) and (10), it becomes clear that $A^{(n)}$ can be built with the entries of the $U^{(i)}$ $(i = 3 : n)$ matrices and the p_i $(i = 1 : n)$, z_i $(i = 2 : n)$ and d_i $(i = 1 : n)$ values. $U^{(i)}$ is the matrix of the left singular vectors of

$$L^{(i)} = \left[\begin{array}{cc} 1 & 0 \\ z_i & p_i \end{array}\right]. \tag{11}$$

The algorithm to compute $A^{(n)}$ (called WE), must start by computing d_i $(i = 1 : n)$, since p_i $(i = 1 : n)$ and z_i $(i = 2 : n)$ depend on d_i; recall that these d_is are a permutation of S^* that can be built using the Lemma 2; the algorithm that performs this permutation is taken from [10].

The WE algorithm can be written as follows:

```
Algorithm Sequential_WE
  1: build d (as mentioned above, taken from [10])
  2: compute p_i (i = 1 : n) and z_i (i = 2 : n), in accordance with (7)
  3: build L^(i) (i = 3 : n), in accordance with (11)
  4: compute SVD(L^(i)) to obtain U^(i) (i = 3 : n), using LAPACK_dgesvd
  5: build A^(n) as shown in (8-10), using BLAS_[dgemm/dscal]
```

It was proved in [10] that the time complexity of the WE Algorithm is

$$T(n) = \left\{n^2 + \frac{328n}{3}\right\} t_f,$$

where t_f is the execution time for a single floating point operation.

Table 1 shows the results of some numerical experiments with the WE algorithm, where S denotes the singular values of the computed lower triangular matrix. In all the cases the results are quite good.

Table 1. Experimental Results (accuracy) with WE algorithm

n	4	5	8	30	50	100	150	300	500
$\|S^* - S\|_2$	4e-16	4e-16	5e-16	1e-15	1e-15	2e-14	4e-14	8e-15	1e-14
$\frac{\|S^* - S\|_2}{\|S^*\|_2}$	1e-16	3e-17	1e-16	1e-16	1e-16	3e-16	2e-16	3e-16	5e-16

3 Parallel algorithm for Distributed Memory model

The tools to implement the distributed memory version were standard linear algebra subroutines and libraries, such as LAPACK [1], BLAS [8] and the communications library BLACS [4] over MPI [7].

Recall that $U^{(i)}$ $(i = 3 : n)$ depends on the SVD of $L^{(i)}$, and $L^{(i)}$ (see (11)) depends on p_i and z_i (defined in (7)). Each element of p can be computed independently of the others; as z_i depends on p_{i-1} and d_i, z_i can also be computed in parallel with z_j $(j = 2 : n,\ j \neq i)$. Then, the SVD of the matrices $L^{(i)}$ can also be computed in parallel and the $U^{(i)}$ matrices can be computed at the same time.

On the other hand, in the former section it was proved that $A \equiv A^{(n)}$ can be computed without explicitly computing $A^{(1)}, A^{(2)}, ..., A^{(n-1)}$. In order to avoid unnecessary floating point operations, we can order the products of each row of A as in the following scheme (suppose the 5-th row of a 6×6 matrix A):

with $U^{(6)}$: $\Rightarrow A_{5,1:6} =$				$u_{11}^{(6)}$	1	
with $U^{(5)}$: $A_{5,4}u_{21}^{(5)} \Rightarrow A_{5,1:6} =$			$u_{11}^{(6)}u_{21}^{(5)}$	$u_{11}^{(6)}$	1	
with $U^{(4)}$: $A_{5,3}u_{21}^{(4)} \Rightarrow A_{5,1:6} =$		$u_{11}^{(6)}u_{21}^{(5)}u_{21}^{(4)}$	$u_{11}^{(6)}u_{21}^{(5)}$	$u_{11}^{(6)}$	1	
with $U^{(3)}$: $A_{5,2}u_{21}^{(3)} \Rightarrow A_{5,1:6} =$	$u_{11}^{(6)}u_{21}^{(5)}u_{21}^{(4)}u_{21}^{(3)}$	$u_{11}^{(6)}u_{21}^{(5)}u_{21}^{(4)}$	$u_{11}^{(6)}u_{21}^{(5)}$	$u_{11}^{(6)}$	1	

The general expressions of this procedure are:

$$A_{i,i-2} = u_{21}^{(i-2)};\quad A_{i,j} = A_{i,j+1}u_{21}^{(j)};\quad (i = n;\ j = i-3, i-2, ..., 1); \tag{12}$$

$$A_{i,i-1} = u_{11}^{(i-1)};\quad A_{i,j} = A_{i,j+1}u_{21}^{(j)};\quad (i = 2 : n-1;\ j = i-2, i-3, ..., 1). \tag{13}$$

To finish the construction of A, the columns of the lower triangular of A, (except the diagonal), are multiplied with the values of z; this operation can be expresed by:

$$A_{ij} = z_j A_{ij};\quad (i = 2 : n;\ j = 1 : i-1). \tag{14}$$

Then, the rows of A can be obtained simultaneously if the values of U and z are available.

Therefore, there are three sections of the WE algorithm amenable for parallelization: the computing of the U matrices, the z components and the A rows. In

this work, the parallelization consists in the distribution of the $n-1$ components of z (z_2,...,z_n), $(n-2)$ U^i matrices ($U^{(3)}$,...,$U^{(n)}$) and $n-1$ rows of A ($A_{2,1:1}$, $A_{3,1:2}$, $A_{4,1:3}$,...,$A_{n,1:n-1}$) among P processors.

To control the distribution of the work among the processors two indexes have been used, called *low* and *up*, which give the limits of the subinterval of components of U and z which each processor must compute.

The distribution of the work needed to obtain A is controlled through the data structures *Rows* and *CountRows*; *Rows* controls which rows of A belongs to each processor, and *CountRows* gives the number of rows in each processor. The distribution of pairs of rows is made trying to equilibrate the computational work. For example, if $n = 20$ and $P = 7$, the following pairs of rows can be formed:

pairs	(2,20)	(3,19)	(4,18)	(5,17)	(6,16)	(7,15)	(8,14)	(9,13)	(10,12)	(11,-)
flops	36	38	38	38	38	38	38	38	38	19

The pairs are formed picking rows from both extremes, so that the total number of flops is approximately the same for every processor. In the example, each processor owns a pair, and the rest of the pairs are distributed among the processors:

Proc	0	1	2	3	4	5	6
Rows	2, 20, 9	3, 19, 13	4, 18, 10	5, 17, 12	6, 16, 11	7, 15	8, 14
CountRows	3	3	3	3	3	2	2

To implement this idea in a distributed memory computer, all the arrays *Rows*, *CountRows*, U and z must be available in all the processors; therefore, the algorithm must contain at least two communication stages. The following diagram outlines how this is scheduled for the case $n = 19$, $P = 4$:

	$Proc_0$	$Proc_1$	$Proc_2$	$Proc_3$
Build	$d_{1:n}$	$d_{1:n}$	$d_{1:n}$	$d_{1:n}$
Compute	$z_{3:7}$	$z_{8:11}$	$z_{12:15}$	$z_{16:19,2}$
Compute	$U^{(3)}...U^{(7)}$	$U^{(8)}...U^{(11)}$	$U^{(12)}...U^{(15)}$	$U^{(16)}...U^{(19)}$
	$\Leftarrow - \Rightarrow$ All–to–All Broadcast of U and z $\Leftarrow - \Rightarrow$			
Compute	$A_{2,1}$, $A_{19,1:18}$ $A_{16,1:15}$, $A_{15,1:14}$ $A_{11,1:10}$	$A_{3,1:2}$, $A_{18,1:17}$ $A_{7,1:6}$, $A_{14,1:13}$ $A_{10,1:9}$	$A_{4,1:3}$, $A_{17,1:16}$ $A_{8,1:7}$, $A_{13,1:12}$	$A_{5,1:4}$, $A_{16,1:15}$ $A_{9,1:8}$, $A_{12,1:11}$
	$\Leftarrow -$ All–to–One communication to build A in $Proc_0$ $\Leftarrow -$			

The algorithm for the distributed computation of z, U and A is described below.

```
Algorithm Parallel_WE
  In Parallel For Proc = 0, 1, ..., P − 1
    1: build d
    2: compute p_i, z_i (i = low : up), in accordance with (7)
    3: build L^(i) (i = low : up), in accordance with (11)
    4: compute SVD(L^(i)) to obtain U^(i) (i = low : up),
                                    using LAPACK_dgesvd
    5: All-to-all broadcast of z and U, using BLACS_dgeb[r/s]2d
    6. compute A_ij (i = Rows_1, Rows_countRows; j = 1 : i − 2),
                                    in accordance with (12-14)
    7: All-to-One Reduction of A to Proc = 0,
                                    using MPI_[Pack/Send/Recv]
  EndParallelFor
```

3.1 Theoretical and Experimental Costs

The code described above has been tested in a cluster of 2GHz biprocessor Intel Xeons (Kefren[4]) composed of 20 nodes, each one with 1 Gbyte of RAM, disposed in a 4×5 mesh with 2D torus topology and interconnected through a SCI network.

The theoretical analysis of the Parallel WE algorithm shows that its speedup is severely affected by the construction of A in the processor $Proc_0$, since the volume of data to be transferred is $O\left(\frac{n^2}{\sqrt{P}}\right)$, as can be seen in the theoretical execution time:

$$T_{WE}(n,P) = \left\{\frac{n^2}{P} + \frac{322n}{3P} + 2n - \frac{656}{3P}\right\} t_f + 5\sqrt{P}t_m + \frac{n^2+4n}{\sqrt{P}}t_v,$$

where t_m is the network latency and t_v is the inverse of the bandwidth; therefore, the WE speedup does not reach the theoretical optimum, according with the following expression:

$$\lim_{n\to\infty} S_{WE}(n,P) = \frac{P}{1+\sqrt{P}\frac{t_v}{t_f}}.$$

Some experiments performed in the Kefren cluster confirm this behaviour. The execution times are recorded in the Table 2, where it is quite clear that the execution times do not decrease when the number of processors increases.

The theoretical cost of the same algorithm without the final construction of A in the processor $Proc_0$ is:

$$T_{WE}(n,P) = \left\{\frac{n^2}{P} + \frac{322n}{3P} + 2n - \frac{656}{3P}\right\} t_f + 4\sqrt{P}t_m + \frac{6n}{\sqrt{P}}t_v,$$

[4] http://www.grycap.upv.es/usuario/kefren.htm

this shows that the degree of parallelism reached during the computation of U, z and A is theoretically good, since the speedup reaches the optimum asymptotically:

$$\lim_{n\to\infty} S_{WE}(n,P) = P.$$

The experiments with the algorithm without building A can be seen in the Table 3. In these experiments there are execution times reductions when we use more than one processor, except in $n = 500$, that is a case efficiently solved by the sequential algorithm. The efficiency of these experiments are in Figure 1. The efficiency curve at $n = \{2000, 2500, 3000\}$ represents a typical case where the use of the processor cache influences the execution times; however, this phenomenon tends to disappear when the problem size increases.

Table 2. WE Execution Times in Kefren, building A in $Proc_0$

P	Seconds							
1	0.41	5.08	12.05	22.28	35.76	66.08	191	270
2	2.61	10.27	21.92	39.31	73.79	125	294	456
4	2.59	10.29	22.02	39.66	72.60	113	258	400
6	2.66	10.60	23.26	42.15	72.71	111	244	388
8	2.81	11.63	26.67	45.55	75.08	114	240	328
10	3.00	11.82	26.90	47.61	77.52	115	237	379
12	3.22	11.90	26.74	47.28	77.27	115	232	375
14	3.51	12.13	27.69	48.67	79.58	118	230	370
16	3.69	12.60	28.51	50.44	81.27	120	230	368
n	500	1000	1500	2000	2500	3000	4000	5000

Table 3. WE Execution Times in Kefren, without building A in $Proc_0$

P	Seconds							
1	0.41	5.08	12.05	22.28	35.76	66.08	191	270
2	2.20	4.68	7.85	12.51	21.30	42.13	104	136
4	1.19	2.56	4.39	7.10	12.23	23.12	53	69
6	0.69	1.89	3.27	5.29	8.96	16.08	35	47
8	0.55	1.55	2.76	4.37	7.18	12.90	27	35
10	0.48	1.39	2.43	3.87	6.21	10.52	21	28
12	0.49	1.26	2.20	3.34	5.43	8.94	18	24
14	0.45	1.18	2.00	3.08	4.92	8.02	16	20
16	0.44	1.10	1.89	2.89	4.49	7.21	14	18
n	500	1000	1500	2000	2500	3000	4000	5000

The reconstruction of a matrix with prescribed singular values is usually a part of a larger problem to be solved in parallel. So, the gathering of the matrix

in a single processor or the redistribution of the matrix among processors may be necessary or not, depending on the larger problem. Therefore, the analysis above shows that, leaving aside the final communications needed to recover the matrix in a single processor, the resolution of the problem has been reasonably parallelized. The remaining pitfall will be adressed in the next section.

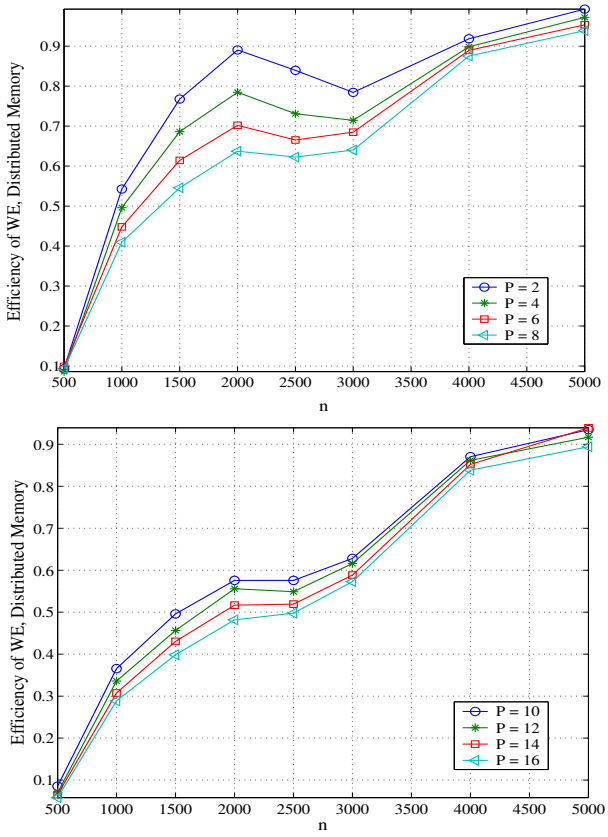

Fig. 1. Experimental WE Efficiency in Kefren, without building A in $Proc_0$

4 Parallel algorithm for Shared Memory model

The analysis of the costs of the distributed memory code shows that the communications needed to collect the results and put it in a single processor damage seriously the performance of the code. In a Shared Memory Machine, this last step would be not needed, so that it is a natural way to improve the overall speed of the code. This implementation has been carried out using OpenMP [2]

compiler directives, such as *omp parallel do* (to parallelize a loop), *omp parallel* (to parallelize a section of code), *omp barrier* (to synchronize execution threads) and *omp do* (to define shared work in a cycle).

Using H process threads and assuming that each one is executed in one processor ($P = H$), each thread shall compute a subset of components of z, U and A; it was seen in the last section that there are no data dependency problems.

The data distribution of z, U and A might be performed by the programmer as in the distributed memory code (through indexes *low*, *up*, *Rows* and *CountRows*). However, in this case it is more efficient to let the openMP compiler do the job; the directives *omp parallel do* and *omp do* split the work among processors automatically. The following diagram shows schematically how would proceed the computation:

$Thread$	Th_0	Th_1	Th_2	Th_3
Build	$d_{1:n}$	– – – – –	– – – – –	– – – – –
Compute	$z_{3:7}$	$z_{8:11}$	$z_{12:15}$	$z_{16:19,2}$
Compute	$U^{(3)}...U^{(7)}$	$U^{(8)}...U^{(11)}$	$U^{(12)}...U^{(15)}$	$U^{(16)}...U^{(19)}$
	⇐ – ⇒ Synchronisation Barrier ⇐ – ⇒			
Compute	$A_{2,1}, A_{3,1:2}, A_{4,1:3}$ $A_{5,1:4}, A_{18,1:17}$	$A_{6,1:5}, A_{7,1:8}$ $A_{8,1:7}, A_{9,1:8}$	$A_{10,1:9}, A_{11,1:10}$ $A_{12,1:11}, A_{13,1:12}$	$A_{14,1:13}, A_{15,1:14}$ $A_{16,1:15}, A_{17,1:16}, A_{19,1:18}$

Comparing this diagram with the Distributed Memory diagram, it is clear that the communication stages disappear, so that the efficiency is expected to increase. This process is written with detail in the ParallelSh_WE algorithm, where we have used *omp parallel* in order to create a team of threads (Th_1, ..., Th_{H-1}), to run in parallel a code segment, and in ParallelSh_A and ParallelSh_zU algorithms, where the directive *omp do* was used to divide the iterations (of the "for" loop) among the the threads created with *omp parallel*.

```
Algorithm ParallelSh_WE
  1: build d /* executed by the main thread */
  2: !$omp parallel private(Th) /* slave threads created by
       3: call ParallelSh_zU                    the main thread */
       4: !$omp barrier
       5: call ParallelSh_A
  6: !$omp end parallel /* slave threads finished by
                                                 the main thread */

Algorithm ParallelSh_A
  1: !$omp do /* The iterations are divided among the threads */
      2: For i = 2, 3, ..., n
        3: compute A_{i,1:i-1}, in accordance with (12-14)
      4: EndFor
  5: !$omp enddo
```

```
Algorithm ParallelSh_zU
  1: !$omp do /* The iterations are divided among the threads */
       2: For i = 3, 4, ..., n
         3: compute z_{i-1}, p_i, in accordance with (7)
         4: build L^(i), in accordance with (11)
         5: compute SVD(L^(i)) to obtain U^(i), using LAPACK_dgesvd
       6: EndFor
  7: !$omp enddo
  8: If Th = Th_{H-1} compute z_1, in accordance with (7)
```

4.1 Experimental Tests

The shared memory code has been tested in a multiprocessor (Aldebaran[5]) SGI Altix 3700 with 48 processors Intel 1.5 GHz Itanium 2, each one with 16 Gbytes of RAM; these are connected with a SGI NumaLink network, with hypercube topology. Although from the programmer's point of view it is a shared memory multiprocessor, actually it is a distributed memory cluster as well, though with a very fast interconnection network. The execution times in this machine are summarized in the Table 4.

The efficiency corresponding to these experiments (Figure 2) is good even with relatively small problem sizes. With $n = 1000$ and two threads the efficiency is 81%; from $n = 2000$ the efficiency is very good with up to 4 threads and acceptable for 6. For the largest case tested in this work ($n = 5000$) the efficiency with 6 threads is also good.

Table 4. WE execution times in Aldebaran (Shared memory)

H	Seconds							
1	1.3	4.9	10.1	22	26	37	68	95
2	1.0	3.1	6.1	12	14	20	35	48
3	0.64	2.1	4.3	9.0	10	14	23	31
4	0.55	1.6	3.4	7.4	7.1	10	18	23
6	0.45	1.1	2.7	5.0	6.2	9	16	18
8	0.44	1.1	2.6	4.3	5.5	8	13	17
10	0.33	1.2	2.3	3.7	4.8	7	10.3	16
12	0.35	1.0	2.6	3.7	4.6	6.1	10.2	13
14	0.32	0.9	2.2	3.3	4.2	6.0	9.8	11
16	0.25	0.8	2.1	3.2	4.1	5.4	9.6	10
n	500	1000	1500	2000	2500	3000	4000	5000

The Scaled Speedup is computed increasing in the same proportion the size of the problem and the number of threads; as WE is $O(n^2)$ we have taken its

[5] http://www.asic.upv.es

Speedup with n ={1000, 1400, 2000, 2800, 3400, 4000} respectively with H ={1, 2, 4, 8, 12, 16} in Figure 3. This figure shows an acceptable scalability.

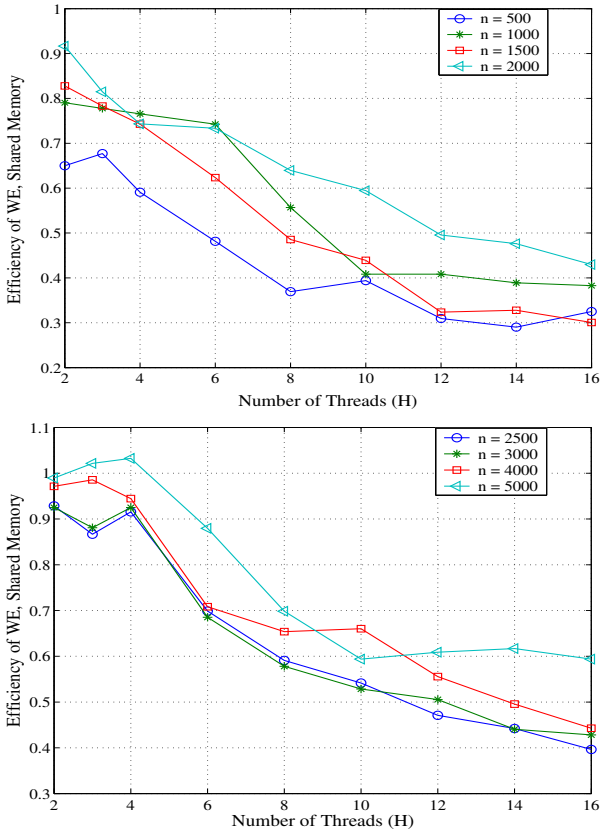

Fig. 2. Experimental WE Efficiency in Aldebaran (Shared memory)

5 Conclusions

The parallel code written for distributed memory reaches a good level of parallelism, as far as the computation phase is concerned. However, once the matrix is computed, it could be necessary to bring it back to a single processor or redistribute it among processors, by depending of the design of the parallel algorithm that uses this matrix; these communications spoil all the gains obtained with the parallel code. This trouble can be overcome if the same algorithm is adapted to a shared memory machine, where these final communications would not be needed. Furthermore, some communications that would happen in the

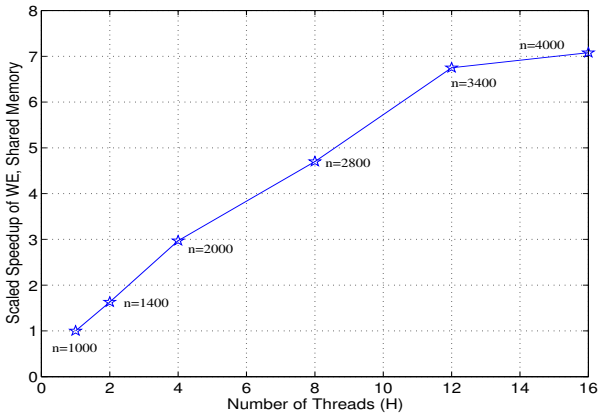

Fig. 3. WE Scaled Speedup in Aldebaran. Shared memory

distributed memory code (replicating U, z) would not be necessary either in the shared memory code.

We can establish the following comparisons between both approaches:

Shared Memory	Distributed Memory
* Easy Implementation	* Complex Implementation
* No extra data structures	* Require extra data structures
* Load Distribution through compiler directives	* Load Distribution by the programmer
* Efficiency larger than 90% with up to 4 threads	* Do not reach acceptable performance due to the last communications stage

Therefore, the shared memory implementation decreases the execution times of the sequential WE code, and gives good performances with up to 4 threads. It reaches an efficiency $> 90\%$ (Figure 2), obtaining as well an acceptable scalability (Figure 3). The performance is damaged when more than 4 threads are used; this is due to the fact that the machine Aldebaran is a multiprocessor with logically shared but physically distributed memory, so that at the end there is a large (transparent to the programmer) traffic of messages.

It seems clear that the nature of this problem makes it more addequate to be processed in a shared memory environment, rather than in a distributed memory cluster.

Acknowledgement

This work has been supported by Spanish MEC and FEDER under Grant TIC2003-08238-C02-02 and SEIT-DGEST-SUPERA-ANUIES (México).

References

1. Anderson E., Bai Z., Bishof C., Demmel J., Dongarra J.: LAPACK User Guide; Second edition. SIAM (1995)
2. Chandra, R., Dagum L., Kohr D., Maydan D., McDonald J., Menon R.: Parallel Programming in OpenMP. Morgan Kaufmann Publishers (2001)
3. Chu, M.T.: Inverse Eigenvalue Problems. SIAM, Review, Vol. 40 (1998)
4. Dongarra J., Van de Geijn R.: Two dimensional basic linear algebra comunications subprograms. Tecnical report st−cs−91−138, Department of Computer Science, University of Tennessee (1991)
5. Flores-Becerra G., García V. M., Vidal A. M.: Numerical Experiments on the Solution of the Inverse Additive Singular Value Problem. Lecture Notes in Computer Science, Vol. 3514, (2005) 17-24
6. Groetsch, C.W.: Inverse Problems. Activities for Undergraduates. The mathematical association of America (1999)
7. Groupp W., Lusk E., Skjellum A.: Using MPI: Portable Parallel Programming with Message Passing Interface. MIT Press (1994)
8. Hammarling S., Dongarra J., Du Croz J., Hanson R.J.: An extended set of fortran basic linear algebra subroutines. ACM Trans. Mathemathical Software (1988)
9. Horn A.: On the eigenvalues of a matrix with prescribed singular values. Proc. Amer. Math. Soc., Vol. 5, (1954) 4-7
10. Kosowski P., Smoktunowicz A.: On Constructing Unit Triangular Matrices with Prescribed Singular Values. Computing, Vol. 64, No. 3 (2000) 279-285
11. Neittaanmki, P., Rudnicki, M., Savini, A.: Inverse Problems and Optimal Design in Electricity and Magnetism. Oxford: Clarendon Press (1996)
12. Sun, N.: Inverse Problems in Groundwater Modeling. Kluwer Academic (1994)