A NOTE ON PARALLEL MATRIX INVERSION*

DEDICATED TO G. W. STEWART ON THE OCCASION OF HIS 60TH BIRTHDAY

ENRIQUE S. QUINTANA[†], GREGORIO QUINTANA[†] , XIAOBAI SUN[‡], AND ROBERT VAN DE GEIJN[§]

Abstract. We present one-sweep parallel algorithms for the inversion of general and symmetric positive definite matrices. The algorithms feature simple programming and performance optimization while maintaining the same arithmetic cost and numerical properties of conventional inversion algorithms. Our experiments on a Cray T3E-600 and a Beowulf cluster demonstrate high performance of implementations for distributed memory parallel computers.

1. Introduction

Despite the inexpedience of matrix inverse, as Higham summarizes in [13], "there are situations in which a matrix inverse must be computed." Examples arise in statistics [4, §7.5], [15, §2.3], [16, pp. 342], [3], in numerical integrations in superconductivity computations [12], and in stable subspace computation in control theory [17]. The recent years have seen increasing interest in parallel solutions of large-scale applications [3,7,9]. Inversion algorithms for general full nonsingular matrices are mostly based on the availability of a complete LU factorization. The algorithm used by LINPACK [8], and LAPACK [2] proceeds as follows.

(1) LU factorization with partial pivoting, $PA = LU$, where P is a permutation matrix, and U, $L \in \mathbb{R}^{n\times n}$ are upper triangular and unit lower triangular matrices, respectively.
(2) Triangular inversion of U (forward substitution).
(3) Triangular (system) solve for X: $XL = U^{-1}$ (backward substitution).
(4) Back permutation of columns, $A^{-1} = XP^{\mathrm{T}}$.

The algorithm allows the inverse to be computed *in-place*, that is, the computed inverse overwrites the input matrix to be inverted. It sweeps three times across the array that houses the involved matrices for LU factorization, triangular inversion, and triangular solve. The algorithm is effective on uni-processor computers and shared memory parallel computers. A parallel version of the algorithm is implemented in SCALAPACK [6]. We present in this paper our study of inversion algorithms via Gauss-Jordan elimination (GJE) for general square matrices and a related algorithm for symmetric positive definite (SPD) matrices. Specifically, we show that these algorithms are more suitable for parallel computers with physically distributed memory.

Matrix inversion using GJE is known to have about the same arithmetic cost as matrix inversion methods using Gaussian elimination (LU factorization) and have a connection with those methods. Many classic references to inversion methods can be found in [13, 14]. Nevertheless, computation arrangements for matrix inversion via GJE (instead of a system solve with multiple right-hand sides) are rarely presented in the literature. A non-blocked parallel version of the in-place GJE based inversion algorithm presented later is given in [10]. The authors of that paper appear not to realize that when pivoting is added the augmented matrix must equal the permutation matrix (a key insight) and a final pivoting of columns is required. An in-place procedure for inversion of positive definite matrices is given by Bauer and Reinsch [5].

In Fig. 1 we describe in MATLAB language a LEVEL-2 BLAS, in-place inversion algorithm via GJE for a general square matrix. Evidently, the sweeps over the intermediate triangular matrices L and U

*THIS PROJECT WAS SUPPORTED IN PART BY THE PRISM PROJECT (ARPA GRANT P-95006) AND A GRANT FROM THE INTEL RESEARCH COUNCIL.

[†]Dept. de Informática, Universidad Jaume I, 12080 Castellón, Spain, {quintana,gquintan}@inf.uji.es.
[‡]Dept. of Computer Science, Duke University, Durham, NC 27708-0129, xiaobai@cs.duke.edu.
[§]Dept. of Computer Sciences, The University of Texas, Austin, TX 78712, rdvg@cs.utexas.edu.

```
% Input  : n x n nonsingular matrix A.
% Output : matrix A overwritten by its inverse
                                         % pivoting
                                         ipivs = [1:n];
for k = 1 : n
                                         [abs_ipiv,ipiv] = max(abs(A(k:n,k)));
                                         ipiv = ipiv+k-1;
                                         [A(k,:),A(ipiv,:)]      = swap(A(ipiv,:),A(k,:));
                                         [ipivs(k),ipivs(ipiv)] = swap(ipivs(ipiv),ipivs(k));
  Akk    = A(k,k);
  % Jordan transformation
  A(:,k) = -[A(1:k-1,k); 0; A(k+1:n,k)]/Akk;
  A      = A + A(:,k) * A(k,:);
  A(k,:) =  [A(k,1:k-1), 1, A(k,k+1:n)]/Akk;
end
                                         A(:,ipivs) = A;
```

Fig. 1: MATLAB code for matrix inversion via GJE. By also executing the steps on the right, pivoting is added.

are computationally eluded. Avoiding the intermediate steps which involve triangular matrices has multiple advantages in parallel computations as we will describe in Sections 5 and 6.

The rest of the paper is organized as follows. In Section 2 we present an inversion algorithm via GJE with partial pivoting. The relation of this algorithm with traditional multi-stage algorithms is given in Section 3. The approach is extended to SPD matrices in Section 4. In Section 5 we discuss parallel implementation issues. In Section 6 we present performance results for our parallel implementations of the algorithm. Concluding remarks can be found in the final section.

2. An Inversion Algorithm via Gauss-Jordan Elimination

Recall how to invert an $n \times n$ matrix A by creating an augmented system $\left(\begin{array}{c|c} A & B \end{array} \right)$ with $B = I_n$, the $n \times n$ identity matrix, and performing Gauss-Jordan elimination on this augmented system:

for $k = 1, n$

$$\textbf{Partition} \left(A \,\|\, B \right) \rightarrow \left(\begin{array}{c|c|c||c|c|c} I_{k-1} & a_{01} & A_{02} & B_{00} & 0 & 0 \\ \hline 0 & \alpha_{11} & a_{12}^T & b_{10}^T & 1 & 0 \\ \hline 0 & a_{21} & A_{22} & B_{20} & 0 & I_{n-k} \end{array} \right)$$

$$\textbf{Update} \left(A \,\|\, B \right) \leftarrow \left(\begin{array}{c|c|c||c|c|c} I_{k-1} & 0 & A_{02} + u_{01} a_{12}^T & B_{00} + u_{01} b_{10}^T & u_{01} = -a_{01}/\alpha_{11} & 0 \\ \hline 0 & 1 & a_{12}^T/\alpha_{11} & b_{10}^T/\alpha_{11} & 1/\alpha_{11} & 0 \\ \hline 0 & 0 & A_{22} + l_{21} a_{12}^T & B_{20} + l_{21} b_{10}^T & l_{21} = -a_{21}/\alpha_{11} & I_{n-k} \end{array} \right)$$

endfor

Magically, upon completion B has been overwritten by A^{-1}.

2.1. An in-place algorithm

After k steps of the above algorithm, it suffices to store only the first k columns of B and the last $n - k$ columns of A. Thus, an algorithm that overwrites A with A^{-1} is given by Upon entering the loop $\left(B_L \,\|\, A_R \right)$ equals A and after its completion it equals A^{-1}.

Partition $A = \left(A_L \,\|\, A_R \right)$ and $B = \left(B_L \,\|\, B_R \right)$ where initially $A_R = A$ and $B_R = I_n$.
for $k = 1, n$

$$\textbf{Partition } \left(B_L \,\|\, A_R \right) \rightarrow \left(\begin{array}{c||c|c} B_{00} & a_{01} & A_{02} \\ \hline b_{10}^T & \alpha_{11} & a_{12}^T \\ \hline B_{20} & a_{21} & A_{22} \end{array} \right) \text{ where } B_{00} \text{ is square.}$$

$$\textbf{Update } \left(B_L \,\|\, A_R \right) \leftarrow \left(\begin{array}{c|c||c} B_{00} + u_{01} b_{10}^T & u_{01} = -a_{01}/\alpha_{11} & A_{02} + u_{01} a_{12}^T \\ \hline b_{10}^T/\alpha_{11} & 1/\alpha_{11} & a_{12}^T/\alpha_{11} \\ \hline B_{20} + l_{21} b_{10}^T & l_{21} = -a_{21}/\alpha_{11} & A_{22} + l_{21} a_{12}^T \end{array} \right)$$

endfor

The above update can also be accomplished by the *rank-1 update*

$$\left(B_L \,\|\, A_R \right) \leftarrow \left(\begin{array}{c|c||c} B_{00} & 0 & A_{02} \\ \hline 0 & 0 & 0 \\ \hline B_{20} & 0 & A_{22} \end{array} \right) + \frac{1}{\alpha_{11}} \left(\begin{array}{c} -a_{01} \\ \hline 1 \\ \hline -a_{21} \end{array} \right) \left(\begin{array}{c|c||c} b_{10}^T & 1 & a_{12}^T \end{array} \right) .$$

The entire computation is in the rank-1 update at a cost of $2n^2$ floating point operations (flops) per iteration for a total cost of $2n^3$ flops. A compact MATLAB code is given in Fig. 1.

2.2. A blocked algorithm

A blocked variant of the above algorithm can be developed in a straight-forward fashion:
Partition $A = \left(A_L \,\|\, A_R \right)$ and $B = \left(B_L \,\|\, B_R \right)$ where initially $A_R = A$ and $B_R = I_n$.
for $k = 1, n$ in steps of b

$$\textbf{Partition } \left(B_L \,\|\, A_R \right) \rightarrow \left(\begin{array}{c||c|c} B_{00} & A_{01} & A_{02} \\ \hline B_{10} & A_{11} & A_{12} \\ \hline B_{20} & A_{21} & A_{22} \end{array} \right) \text{ where } B_{00} \text{ is square and } A_{11} \text{ is } b \times b.$$

Update

$$\left(B_L \,\|\, A_R \right) \leftarrow \left(\begin{array}{c|c||c} B_{00} & 0 & A_{02} \\ \hline 0 & 0 & 0 \\ \hline B_{20} & 0 & A_{22} \end{array} \right) + \left(\begin{array}{c} -A_{01} A_{11}^{-1} \\ \hline A_{11}^{-1} \\ \hline -A_{21} A_{11}^{-1} \end{array} \right) \left(\begin{array}{c|c||c} B_{10} & I & A_{21} \end{array} \right)$$

endfor

The benefit of the blocked algorithm is that most of the computation is now in a matrix-matrix multiply (rank-k update) which allows high performance to be achieved in a portable fashion.

2.3. Adding pivoting

As for the LU factorization, in order to improve stability for general square nonsingular matrices it is necessary to include row pivoting in the algorithm. To add pivoting to our nonblocked algorithm, before the update step, the row with the largest absolute value among elements of α_{11} and a_{21} is swapped with the k-th row. If the augmented system is taken to equal $\left(A \,\|\, B \right)$ with $B = P^T$ instead, we can guarantee that after $k-1$ steps *and after swapping the rows for the current step* the system looks like

$$\left(\begin{array}{c|c|c||c|c|c} I_{k-1} & a_{01} & A_{02} & B_{00} & 0 & 0 \\ \hline 0 & \alpha_{11} & a_{12}^T & b_{10}^T & 1 & 0 \\ \hline 0 & a_{21} & A_{22} & B_{20} & 0 & P_{n-k} \end{array} \right),$$

where P_{n-k} is some permutation matrix determined by future iterations of the loop. The usual update step then proceeds. This time, $A^{-1}P^T$ is computed, so that it is necessary to complete the process by permuting the columns of the result: $A^{-1} = (A^{-1}P^T)P$. Fig. 1 shows how pivoting can be added to the MATLAB code.

Likewise, pivoting is added to the above blocked algorithm by changing the body of the loop to

Partition (as before)

Compute $P\left(\frac{A_{11}}{A_{21}}\right) \to \left(\frac{L_{11}}{L_{21}}\right) \; U_{11}$ (LU factorization with pivoting)

Swap $\left(\begin{array}{c||c|c} B_{10} & A_{11} & A_{12} \\ \hline B_{20} & A_{21} & A_{22} \end{array}\right) \leftarrow P \left(\begin{array}{c||c|c} B_{10} & A_{11} & A_{12} \\ \hline B_{20} & A_{21} & A_{22} \end{array}\right)$

Update (as before)

There are a number of ways to compute the update. As for the unblocked algorithm, a final permutation of the columns is required.

The update in the blocked algorithm is equivalent to the update

$$(1) \qquad \left(\begin{array}{c||c} B_L & A_R \end{array} \right) \leftarrow \left(\begin{array}{c|c||c} B_{00} & 0 & A_{02} \\ \hline 0 & 0 & 0 \\ \hline B_{20} & 0 & A_{22} \end{array}\right) + \left(\begin{array}{c} -A_{01}U_{11}^{-1} \\ \hline U_{11}^{-1} \\ \hline -L_{21} \end{array}\right) \left(\begin{array}{c|c||c} L_{11}^{-1}B_{10} & L_{11}^{-1} & L_{11}^{-1}A_{21} \end{array} \right)$$

where L_{11}, L_{21} and U_{11} are as computed by the LU factorization with partial pivoting of the corresponding part of A. Here $-A_{01}U_{11}^{-1}$, $L_{11}^{-1}B_{10}$, and $L_{11}^{-1}A_{21}$ are computed using triangular solves rather than matrix inversion. Forming $A_{11}^{-1} = U_{11}^{-1}L_{11}^{-1}$ is accomplished by the last two stages of the traditional algorithm. Computing the update in this fashion has the benefit that since explicit use of A_{11}^{-1} is avoided, stability of the algorithm should be similar to that exhibited by the traditional approach. More on this in the next section.

3. Relation with multi-stage algorithms

The explicit use of A_{11}^{-1} in the blocked algorithm with pivoting should raise eyebrows. In [18], we show that the numerical stability of the overall algorithm is only mildly affected by the use of the inverse, with stability deteriorating as a function of the algorithmic block size b. By instead using the variant that updates the matrix given in (1) these instabilities can be avoided.

Consider yet another alternative for computing A^{-1}:

(1) LU factorization with partial pivoting, $PA = LU$.
(2) Triangular inversion of U.
(3) Triangular inversion of L.
(4) Multiplication of triangular matrices $X = U^{-1}L^{-1}$.
(5) Back permutation of columns, $A^{-1} = XP^{\mathrm{T}}$.

For simplicity we ignore pivoting and give blocked algorithms for the four remaining stages in Fig. 2. If one picks the block size b to be equal at each step of the algorithms, it becomes clear that all four algorithms can be implemented by merging the loops. Critical to this observation is the fact that the different algorithms update four different quadrants of the matrix. Once the four loops have been merged into one loop, a number of intermediate computations can be eliminated leaving only the computations in the boxes in Fig. 2. **Careful examination of these computations show that they are exactly the updates of the nine submatrices as given in (1).** The same conclusion holds when pivoting is added.

We conclude that the algorithm based on (1) is equivalent to this multi-stage algorithm and thus shares its stability properties. In [13] it is shown that the traditional approach and this alternative multi-stage algorithm are essentially equally stable.

Factor $A \leftarrow L\backslash U = \mathrm{LU}(A)$:

Partition $A = \left(\begin{array}{c||c} A_{UL} & A_{UR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right)$
where A_{UL} is 0×0
do until A_{BR} is 0×0
Determine block size b
View

$$\left(\begin{array}{c||c} A_{UL} & A_{UR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c||c|c} L\backslash U_{00} & U_{01} & U_{02} \\ \hline\hline L_{10} & A_{11} & A_{12} \\ \hline L_{20} & A_{21} & A_{22} \end{array}\right)$$

where A_{11} is $b \times b$

$A_{11} \leftarrow L\backslash U_{11} = \mathrm{LU}(A_{11})$
$A_{21} \leftarrow L_{21} \quad = A_{21} U_{11}^{-1}$
$A_{12} \leftarrow U_{12} \quad = L_{11}^{-1} A_{12}$
$A_{22} \leftarrow \hat{A}_{22} \quad = A_{22} - L_{21} U_{12}$
$\quad (= \mathbf{A_{22}} - (\mathbf{A_{21} U_{11}^{-1}})(\mathbf{L_{11}^{-1} A_{12}}))$

Continue with

$$\left(\begin{array}{c||c} A_{UL} & A_{UR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c|c||c} L\backslash U_{00} & U_{01} & U_{02} \\ \hline L_{10} & \mathbf{L\backslash U_{11}} & \mathbf{U_{12}} \\ \hline\hline L_{20} & \mathbf{L_{21}} & \mathbf{\hat{A}_{22}} \end{array}\right)$$

enddo

Invert $U \leftarrow \hat{U} = U^{-1}$:

Partition $U = \left(\begin{array}{c||c} U_{UL} & U_{UR} \\ \hline\hline 0 & U_{BR} \end{array}\right)$
where U_{UL} is 0×0
do until U_{BR} is 0×0
Determine block size b
View

$$\left(\begin{array}{c||c} U_{UL} & U_{UR} \\ \hline\hline 0 & U_{BR} \end{array}\right) = \left(\begin{array}{c||c|c} \hat{U}_{00} & U_{01} & U_{02} \\ \hline\hline 0 & U_{11} & U_{12} \\ \hline 0 & 0 & U_{22} \end{array}\right)$$

where U_{11} is $b \times b$

$U_{11} \leftarrow \hat{U}_{11} = U_{11}^{-1}$
$U_{01} \leftarrow \hat{U}_{01} = U_{01} U_{11}^{-1}$
$U_{02} \leftarrow U_{02}^{\star} = U_{02} - \hat{U}_{01} U_{12}$
$\quad (= \mathbf{U_{02}} - (\mathbf{U_{01} U_{11}^{-1}})(\mathbf{L_{11}^{-1} A_{12}}))$
$U_{12} \leftarrow U_{12}^{\star} = -U_{11}^{-1} U_{12}$
$\quad (= -\mathbf{U_{11}^{-1}}(\mathbf{L_{11}^{-1} A_{12}}))$

Continue with

$$\left(\begin{array}{c||c} U_{UL} & U_{UR} \\ \hline\hline 0 & U_{BR} \end{array}\right) = \left(\begin{array}{c|c||c} \hat{U}_{00} & \mathbf{\hat{U}_{01}} & \mathbf{U_{02}^{\star}} \\ \hline 0 & \mathbf{\hat{U}_{11}} & \mathbf{U_{12}^{\star}} \\ \hline\hline 0 & 0 & U_{22} \end{array}\right)$$

enddo

Invert $L \leftarrow \hat{L} = L^{-1}$:

Partition $L = \left(\begin{array}{c||c} L_{UL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right)$
where L_{UL} is 0×0
do until L_{BR} is 0×0
Determine block size b
View

$$\left(\begin{array}{c||c} L_{UL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) = \left(\begin{array}{c||c|c} \hat{L}_{00} & 0 & 0 \\ \hline\hline L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right)$$

where L_{11} is $b \times b$

$L_{11} \leftarrow \hat{L}_{11} = L_{11}^{-1}$
$L_{10} \leftarrow \hat{L}_{10} = L_{11}^{-1} L_{10}$
$L_{20} \leftarrow L_{20}^{\star} = L_{20} - L_{21} \hat{L}_{10}$
$\quad (= \mathbf{L_{20}} - (\mathbf{A_{21} U_{11}^{-1}})(\mathbf{L_{11}^{-1} L_{10}}))$
$L_{21} \leftarrow L_{21}^{\star} = -L_{21} L_{11}^{-1}$
$\quad (= -(\mathbf{A_{21} U_{11}^{-1}})\mathbf{L_{11}^{-1}})$

Continue with

$$\left(\begin{array}{c||c} L_{UL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) = \left(\begin{array}{c|c||c} \hat{L}_{00} & 0 & 0 \\ \hline \mathbf{\hat{L}_{10}} & \mathbf{\hat{L}_{11}} & 0 \\ \hline\hline \mathbf{L_{20}^{\star}} & \mathbf{L_{21}^{\star}} & L_{22} \end{array}\right)$$

enddo

Compute $\hat{A} \leftarrow \hat{U}\hat{L} = U^{-1}L^{-1} = A^{-1}$:

Partition $\hat{L}\backslash\hat{U} = \left(\begin{array}{c||c} A_{UL} & A_{UR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right)$
where A_{UL} is 0×0
do until A_{BR} is 0×0
Determine block size b
View

$$\left(\begin{array}{c||c} A_{UL} & A_{UR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c||c|c} \hat{A}_{00} & \hat{U}_{01} & \hat{U}_{02} \\ \hline\hline \hat{L}_{10} & \hat{L}\backslash\hat{U}_{11} & \hat{U}_{12} \\ \hline \hat{L}_{20} & \hat{L}_{21} & \hat{L}\backslash\hat{U}_{22} \end{array}\right)$$

where $\hat{L}\backslash\hat{U}_{11}$ is $b \times b$

$\hat{A}_{00} \leftarrow \hat{A}_{00}^{\star} \quad = \hat{A}_{00} - \hat{U}_{01}\hat{L}_{10}$
$\quad (= \mathbf{\hat{A}_{00}} - (\mathbf{\hat{U}_{01} U_{11}^{-1}})(\mathbf{L_{11}^{-1} L_{10}}))$
$\hat{L}_{10} \leftarrow \hat{A}_{10}^{\star} \quad = \hat{U}_{11}\hat{L}_{10}$
$\quad (= \mathbf{U_{11}^{-1}}(\mathbf{L_{11}^{-1} L_{10}}))$
$\hat{U}_{01} \leftarrow \hat{A}_{01}^{\star} \quad = \hat{U}_{01}\hat{L}_{11}$
$\quad (= (\mathbf{U_{01} U_{11}^{-1}})\mathbf{L_{11}^{-1}})$
$\hat{L}\backslash\hat{U}_{11} \leftarrow \hat{A}_{11}^{\star} = \hat{U}_{11}\hat{L}_{11}$
$\quad (= \mathbf{U_{11}^{-1} L_{11}^{-1}})$

Continue with

$$\left(\begin{array}{c||c} A_{UL} & A_{UR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c|c||c} \mathbf{\hat{A}_{00}^{\star}} & \mathbf{\hat{A}_{01}^{\star}} & \hat{U}_{02} \\ \hline \mathbf{\hat{A}_{10}^{\star}} & \mathbf{\hat{A}_{11}^{\star}} & \hat{U}_{12} \\ \hline\hline \hat{L}_{20} & \hat{L}_{21} & \hat{L}\backslash\hat{U}_{22} \end{array}\right)$$

enddo

Fig. 2: Blocked algorithms for a four-stage algorithm to compute A^{-1}: $A \rightarrow LU$, $\hat{L} \leftarrow L^{-1}$, $\hat{U} \leftarrow U^{-1}$, and $A^{-1} = \hat{U}\hat{L}$.

4. Symmetric Positive Definite Matrix Inversion

The techniques described above can be used to derive a blocked algorithm for symmetric positive definite matrix inversion.

Let A be symmetric positive definite. Then its inverse can be computed by first computing its Cholesky factor L, where $A = LL^T$, after which $A^{-1} = {L^{-1}}^T L^{-1}$. Naturally, this can be accomplished in three stages: (1) Compute the Cholesky factorization, (2) Invert lower triangular matrix L, and (3) Multiply $A^{-1} = {L^{-1}}^T L^{-1}$. By putting all these stages together, a one-sweep algorithm can be derived: Partition the current matrix $A^{(k)}$, the original matrix A, and the factor L like

$$A^{(k)} = \left(\begin{array}{c||c} A_{TL}^{(k)} & \star \\ \hline\hline A_{BL}^{(k)} & A_{BR}^{(k)} \end{array} \right), A = \left(\begin{array}{c||c} A_{TL} & \star \\ \hline\hline A_{BL} & A_{BR} \end{array} \right), \text{ and } L = \left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array} \right),$$

where the $\star$ indicated the symmetric part not to be updated and $A_{TL}^{(k)}$, A_{TL}, and L_{TL} are $k \times k$. Repartition

$$A^{(k)} = \left(\begin{array}{c||c} A_{TL}^{(k)} & \star \\ \hline\hline A_{BL}^{(k)} & A_{BR}^{(k)} \end{array} \right) = \left(\begin{array}{c||c|c} A_{00} & \star & \star \\ \hline\hline A_{10} & A_{11} & \star \\ \hline A_{20} & A_{21} & A_{32} \end{array} \right),$$

and consider the update

$$A^{(k+b)} = \left(\begin{array}{c||c} A_{TL}^{(k+b)} & \star \\ \hline\hline A_{BL}^{(k+b)} & A_{BR}^{(k+b)} \end{array} \right) = \left(\begin{array}{c|c||c} A_{00} + \hat{L}_{10}^T \hat{L}_{10} & \star & \star \\ \hline \hat{L}_{10} & L_{11}^{-T} L_{11}^{-1} & \star \\ \hline\hline A_{20} - L_{21} \hat{L}_{10} & L_{21} & A_{32} - L_{21} L_{21}^T \end{array} \right),$$

where $L_{21} = A_{21} L_{11}^{-1}$ and $\hat{L}_{10} = L_{11}^{-1} A_{10}$. This update maintains the state

$$A^{(k)} = \left(\begin{array}{c||c} A_{TL}^{(k)} & \star \\ \hline\hline A_{BL}^{(k)} & A_{BR}^{(k)} \end{array} \right) = \left(\begin{array}{c||c} L_{TL}^{-T} L_{TL}^{-1} & \star \\ \hline\hline L_{BL} L_{TL}^{-1} & A_{BR} - L_{BL} L_{BL}^T \end{array} \right).$$

Notice that once $k = n$, $A^{(k)} = A^{-1}$, the desired result.

5. Parallel implementation

Sadly, the above described algorithms yield no real performance benefits on a sequential computer. The primary reason is that on sequential architectures the invidual stages of the traditional algorithm lend themselves well to optimization. On distributed memory parallel architectures, the story is different.

All three stages of the traditional approach suffer from load-imbalance when executed on distributed memory parallel architectures: During the LU factorization, the active part of the matrix shrinks from an $n \times n$ matrix to a 1×1 matrix as the computation unfolds. For the inversion of U, initially the active matrix is 1×1, eventually expanding to $n \times n$. In addition, the computation only acts on the upper-triangular portion of the matrix. During the computation of the solution X to $XL = U^{-1}$, the active matrix expands from $1 \times n$ to $n \times n$. While cyclic wrapping of the matrices helps load-balance, there is still a considerable performance hit. Similar issues affect the three stages for inverting a symmetric positive definite matrix.

The blocked algorithm presented in the previous section does not suffer the same fate: The bulk of the computation is in the rank-k update given in Section 2.2 which parallelizes almost perfectly.

Indeed, the computation required for the GJE based inversion algorithm is almost identical to the sequence of rank-k updates used to implement highly efficient parallel matrix-matrix multiplication algorithms [1, 11, 19]. Cyclic wrapping of the matrix is not even necessary for load-balance! Even when the inversion is part of a larger computation and wrapping is desirable, excellent performance can be achieved, as we will show in the next section. For inverting a SPD matrix, wrapping is still needed since only the lower triangular part of the matrix is updated.

6. Experimental Results

We present results for three parallel implementations of algorithms for inversion of a general matrix and one for inversion of a SPD matrix. `SL_IGEP` is part of ScaLAPACK [6] and implements the traditional inversion algorithm via LU factorization; `SL_IGJEP` was coded using ScaLAPACK parallel BLAS kernels (PBLAS). It implements a blocked GJE algorithm with the algorithmic block size limited by the distribution block size. All operations required to update the current column panel are performed within a single column of processors. A second implementation of the blocked GJE algorithm, `PLA_IGJEP`, was coded using PLAPACK [20]. In this implementation the current column panel is redistributed so that all processors participate when the unblocked algorithm is used to update that panel. This results in better load balance during this stage of the algorithm and simplifies the coding when the algorithmic block size is to be larger than the distribution block size. While `SL_IGJEP` represents a basic implementation of the algorithm, `PLA_IGJEP` includes a number of optimizations, some of which cannot be (easily) implemented using ScaLAPACK. `PLA_ISPD` implements a parallel blocked one-sweep algorithm for the inversion of SPD matrices.

Performance results are given for two different distributed memory architectures using 64-bit real arithmetic: The Cray T3E-600 (300 MHz) and a Beowulf cluster. The Cray T3E-600 consists of DEC Alpha EV5 with 128 Mbytes of RAM each, interconnected via a three-dimensional toroidal interconnect. The Beowulf cluster consists of Intel Pentium-II (300 MHz) processors running on a 66 MHz Intel Atlanta motherboard with 128MBbytes of RAM each, and interconnected via a 1Gbps Myrinet swicht.

We used ScaLAPACK available from netlib (version 1.6) with the MPIBLACS precompiled for Cray T3E* and Linux. The PLAPACK version, implemented using an alpha release of PLAPACK R2.0, is also MPI based. Optimized sequential Basic Linear Algebra Subprograms (BLAS) were used to attain high performance locally on each processor. The sequential BLAS routine `gemm` delivers 400–450 Mflop/s (millions of flops per second) on a single processor of the Cray T3E and 175–200 Mflop/s on one Pentium-II (300 MHz) processor. We report performance of the inverse routines measuring Mflop/s per processor using the established operation count of $2n^3$ flops for inversion of a general $n \times n$ matrix and n^3 flops for inversion of a SPD matrix.

In Fig. 3, we show that very respectable performance is attained on both architectures using 32 processors. Algorithmic and distribution block sizes of 48 and 32 were used for `SL_IGEP` and `SL_IGJEP` on the T3E and cluster, respectively. As expected, `SL_IGJEP` outperforms `SL_IGEP` since it incurs less communication and exhibits better load balance. For `PLA_IGJEP` we report performance when algorithmic and distribution block sizes are both equal to those used for ScaLAPACK (`PLA_IGJEP_48_48` and `PLA_IGJEP_32_32`). We also show that better performance is attained when a smaller distribution block size and larger algorithmic block size is used (`PLA_IGJEP_24_96` and `PLA_IGJEP_16_64`). The larger algorithmic block size allows local matrix-matrix multiplication to attain higher performance, which explains the asymptotically better performance attained by `PLA_IGJEP_24_96`. The smaller distribution block size yields better load-balance. For smaller problems, the additional cost

*The Cray Scientific library provides a version of ScaLAPACK that uses the *shmem* library for communication. For a fair comparison between `PLA_IGJEP` and `SL_IGJEP`, we chose the MPI based version of ScaLAPACK. Only for small matrices is the shmem based version of ScaLAPACK noticeably faster than the MPI based version.

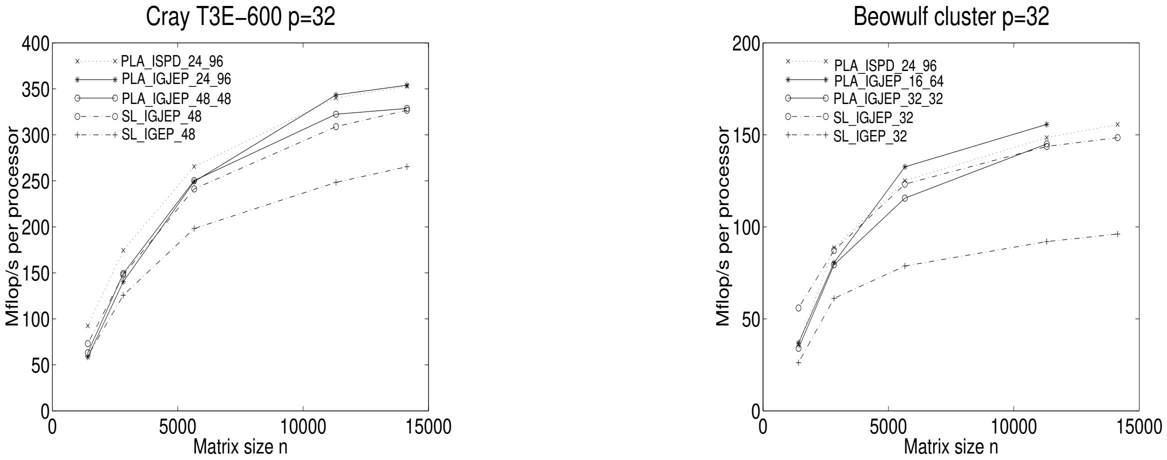

Fig. 3: Mflop/s per processor on the Cray T3E-600 (left) and the Beowulf cluster (right).

of redistributing the current column panel results in a reduction in performance relative to the ScaLAPACK implementation. Finally, in that figure we report performance of the SPD inversion routine using an algorithmic block size of 96/64 and distribution block size of 24/16 (**PLA_ISPD_24_96** and **PLA_ISPD_16_64**). In Fig. 3 (right) we present similar performance results for the Beowulf cluster.

We do not present scalability results for these algorithms since scalability is clearly at least as good at that of parallel implementations for the invidual stages of the traditional algorithm, which themselves are known to have very good scalability properties. For details, see [18].

7. Concluding Remarks

We have described matrix inversion via Gauss-Jordan elimination for general matrices and a related algorithm for SPD matrices. The Gauss-Jordan approach presents the same arithmetic cost as the conventional inversion algorithm while maintaining its numerical properties. Our matrix inversion algorithms render simple programming and performance optimization, which are specially appropriate for parallel computers with distributed memory. The experimental results on a Cray T3E-600 and a Beowulf cluster show that very high performance is attained by the parallel Gauss-Jordan inversion algorithm for general matrices and the parallel one-sweep inversion algorithm for SPD matrices.

Implementations are available at `http://www.cs.utexas.edu/users/plapack/inverse/`.

Acknowledgments.

We express our gratitude to Y. J. Wu and N. Higham for their valuable suggestions.

Access to equipment was provided by the National Partnership for Advanced Computational Infrastructure (NPACI), The University of Texas Advanced Computing Center (TACC), and the Earth and Space Science (ESS) Project of the NASA HPCC Program.

REFERENCES

[1] R. C. Agarwal, F. Gustavson, and M. Zubair. A high-performance matrix multiplication algorithm on a distributed memory parallel computer using overlapped communication. *IBM J. Research and Development*, 38(6), 1994.
[2] E. Anderson, Z. Bai, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. E. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK User's Guide*. SIAM, 1995.

[3] G. Baker. *Implementation of parallel processing to selected problems in satellite geodesy.* PhD thesis, Department of Aerospace Engineering, The Univ. of Texas, 1997.

[4] Y. Bard. *Nonlinear parameter estimation.* Academic Press, New York, NY, 1974.

[5] F. L. Bauer and C. Reinsch. Inversion of positive definite matrices by the Gauss-Jordan method. In J. H. Wilkinson and C. Reinsch, editors, *Linear Algebra, volume II of Handbook for Automatic Computation*, pages 45–49. Springer-Verlag, Berlin, 1971. Contribution I/3.

[6] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK User's Guide.* SIAM, 1997.

[7] T. Cwik, R. van de Geijn, and J. Patterson. The application of parallel computation to integral equation models of electromagnetic scattering. *J. of the Optical Society of America*, 11(4):1538–1545, 1994.

[8] J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart. *LINPACK User's Guide.* SIAM, 1979.

[9] J.D. Gardiner and A.J. Laub. A generalization of the matrix-sign-function solution for algebraic Riccati equations. *Int. J. of Control*, 44:823–832, 1986.

[10] A. V. Gerbessiotis. Algorithmic and practical considerations for dense matrix computations on the BSP model. Technical Report PRG-TR-32-97, Oxford University Computing Laboratory, Oxford, UK, October 1997.

[11] J. Gunnels, C. Lin, G. Morrow, and R. van de Geijn. A flexible class of parallel matrix multiplication algorithms. In *Proceedings of First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing (1998 IPPS/SPDP '98)*, pages 110–116, 1998.

[12] M. T. Heath, G. A. Geist, and J. B. Drake. Early experience with the Intel iPSC/860 at ORNL. Technical Report ORNL/TM-11655, Oak Ridge National Laboratory, TN, USA, September 1990.

[13] N. J. Higham. *Accuracy and stability of numerical algorithms.* SIAM, 1996.

[14] A. S. Householder. *The theory of matrices in numerical analysis.* Dover Publications, 1974.

[15] J. H. Maindonald. *Statistical computation.* Wiley, New York, NY, 1984.

[16] P. McCullagh and J. A. Nelder. *Generalized linear models.* Chapman and Hall, London, 1989.

[17] J. Roberts. Linear model reduction and solution of algebraic Riccati equations by the use of the sign function. *Int. J. Control*, 32:677–687, 1980.

[18] X. Sun, E. S. Quintana, G. Quintana, and R. van de Geijn. Efficient matrix inversion via Gauss-Jordan elimination and its parallelization. Technical Report CS-98-19, Dept. of Computer Sciences, The Univ. of Texas, 1998. http://www.cs.utexas.edu/users/plapack/papers.

[19] R. van de Geijn and J. Watts. SUMMA: Scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience*, to appear.

[20] R. A. van de Geijn. *Using PLAPACK: Parallel Linear Algebra Package.* The MIT Press, 1997.