Efficient Matrix Inversion via Gauss-Jordan Elimination and its Parallelization

Enrique S. Quintana* Gregorio Quintana† Xiaobai Sun‡
Robert van de Geijn§

August, 1998

Abstract. We present a new parallel matrix inversion algorithm and report its implementation on parallel computers with distributed memory. The algorithm features natural load balance, simple programming and easy performance optimization, while maintaining the same arithmetic cost and numerical properties of the conventional inversion algorithm. Our analysis and experiments on a CRAY T3E report near-peak performance for the new approach.

1 Introduction

Despite the inexpedience of matrix inverse, as Higham summarizes in [12], "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 [11], and in stable subspace computation in control theory [18]. The recent years have seen increasing interest in parallel solutions of large-scale applications [2,3,7,14,10]. Inversion algorithms for general full nonsingular matrices are mostly based on the availability of a complete LU factorization. The algorithm used by LINPACK XGEDI [8], and LAPACK XGETRI [1] proceeds as follows.

*Departamento de Informática, Universidad Jaime I, 12071 Castellón, Spain, quintana@inf.uji.es.
†Same address as second author, gquintan@inf.uji.es.
‡Department of Computer Science, Duke University, D107, Levine Science Research Center, Durham, NC 27708-0129, xiaobai@cs.duke.edu.
§Department of Computer Science, The University of Texas, Taylor Hall 2.124, Austin, TX 78712, rdvg@cs.utexas.edu.

(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). Specifically, we show that a one-sweep inversion algorithm via GJE is more suitable on 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 [12, 13]. Nevertheless, computation arrangements for matrix inversion via GJE (instead of a system solve with multiple right hands) are rarely presented in the literature. An in-place procedure for inversion of positive definite matrices is given by Bauer and Reinsch [5]. In Figure 1 we describe in Matlab language a Level-2 blas,

```
% Input : A ∈ IR^{n×n} nonsingular.
% Output : A^{-1} ∈ IR^{n×n} overwritten onto A.
for k = 1 : n
    pivot  = A(k,k);
    A(:,k) = -[A(1:k-1,k); 0; A(k+1:n,k)] / pivot;   % column scaling
    A      = A + A(:,k) * A(k,:);                      % rank-1 update
    A(k,:) = [A(k,1:k-1), 1, A(k,k+1:n)] / pivot;      % row scaling
end
```

Figure 1: The merged algorithm for matrix inversion (without pivoting)

in-place inversion algorithm via GJE for a general square matrix with all of its leading submatrices nonsingular. Evidently, it is a one-sweep approach as opposed to the conventional three-sweep approach. In other words, the sweeps over the intermediate triangular matrices L and U are computationally eluded. Avoiding the intermediate steps which involve non-square (non-rectangular) matrices has multiple advantages in parallel computations as we will describe in Section 5.3.

The rest of the paper is organized as follows. In Section 2 we present a one-sweep inversion algorithm via GJE with partial pivoting. In Section 3 we describe the connection and differences between matrix inversion with GJE and Gaussian elimination in numerical computation. In Section 4, we present an error analysis framework for various inversion methods that can be connected to the LU factorization. This unification in error analysis for inversion algorithms is new to the best knowledge of the authors. In Section 5 we provide several variants of the inversion algorithms via GJE for different computing environments and discuss parallel algorithm development issues. In Sections 6 and 7 we present respectively the performance model and the experimental results from our parallel implementation of the algorithm.

We provide algorithm prototypes in MATLAB, with occasional minor changes in notation to avoid unnecessary details. We denote by e_j the the j-th column of the identity matrix of order n, I_n. We denote by $E_{j|}$ and $E_{|j}$ the first j columns of E and the last j columns of a matrix E, respectively.

2 An Inversion Algorithm via GJE with Partial Pivoting

We describe in this section an inversion algorithm via GJE with partial pivoting (IGJEP). We first review the Gauss-Jordan elimination process. Denote by $J(k, v_k, \delta_k)$, or simply, J_k, a nonsingular matrix having the structure

$$\begin{aligned} J(k, v_k, \delta_k) &= D_k^{-1}(I_n - v_k e_k^{\mathrm{T}}) = D_k^{-1} - v_k e_k^{\mathrm{T}}, \quad e_k^{\mathrm{T}} v_k = 0, \\ D_k &= \mathrm{diag}(I_{k-1}, \delta_k, I_{n-k}), \quad \delta_k \neq 0. \end{aligned} \tag{1}$$

In other words, J_k is an elementary Gauss-Jordan reduction matrix of index k (see [13, 19]) scaled by δ_k^{-1} in row k. For convenience, we call J_k a *scaled Jordan matrix* and the combined vector $g_k = \delta_k^{-1} e_k - v_k$ the *Jordan vector* associated with J_k. Given a vector $u = (\mu_1, \ldots, \mu_n)^{\mathrm{T}} \in I\!R^n$, $u \not\equiv 0$; for $k = 1{:}n$, there exist a permutation matrix P_k of order n and a scaled Jordan matrix J_k such that $J_k(P_k u) = e_k$. Here, P_k can be a permutation matrix swapping μ_k and $\mu_{\bar{k}}$ such that $\mu_{\bar{k}} \neq 0$, and J_k is determined by

$$v_k = (P_k u)/\mu_{\bar{k}} - e_k \quad \text{and} \quad \delta_k = \mu_{\bar{k}}. \tag{2}$$

```
% Input : A ∈ IR^{n×n} nonsingular.
% Output : A^{-1} ∈ IR^{n×n} overwritten onto A.
ipiv = [1 : n];
for k = 1 : n
    % pivoting
    [ abspivot, k̄ ] = max( [ zeros(k−1, 1); abs(A(k:n, k)) ] );
    pivot = A(k̄, k);
    [ A(k, :), A(k̄, :) ] = swap( A(k, :), A(k̄, :) );
    [ ipiv(k), ipiv(k̄) ] = swap( ipiv(k), ipiv(k̄) );
    % Jordan transformation
    A(:, k)  = −[ A(1:k−1, k); 0; A(k+1:n, k) ] / pivot;
    A        = A + A(:, k) ∗ A(k, :);
    A(k, :)  = [ A(k, 1:k−1), 1, A(k, k+1:n) ] / pivot;
end
A(:, ipiv) = A;        % backward permutation
```

Figure 2: The merged algorithm for matrix inversion with pivoting (MAMIP)

We now describe and verify algorithm IGJEP. Let $A^{(0)} = A$, $B^{(0)} = I_n$, $\Pi_0 = I_n$, and $G_0 = [\,]$ (an empty matrix). At the beginning of step k of IGJEP, $k = 1{:}n$, $A^{(k-1)}$ and $B^{(k-1)}$ are mathematically related by

$$A^{(k-1)} = B^{(k-1)} A,$$

and the matrices $A^{(k-1)}$ and $B^{(k-1)}$ have the following structure,

$$\begin{aligned} A^{(k-1)} &= [\, I_{k-1|}, A^{(k-1)}_{|n-k+1} \,], \\ B^{(k-1)} &= [\, G_{k-1}, I_{|n-k+1} \,]\, \Pi_{k-1}, \end{aligned} \tag{3}$$

where Π_{k-1} is a permutation matrix. Let $a_k = A^{(k-1)} e_k = (\alpha_1, \ldots, \alpha_n)^{\mathrm{T}}$. Since A is nonsingular, $\alpha_{\bar{k}} \neq 0$, for some $\bar{k}$, $k \leq \bar{k} \leq n$. Determine P_k that swaps α_k and $\alpha_{\bar{k}}$, and J_k such that $J_k(P_k a_k) = e_k$. Let $A^{(k)} = J_k P_k A^{(k-1)}$, $B^{(k)} = J_k P_k B^{(k-1)}$, and $\Pi_k = P_k \Pi_{k-1}$. Then,

$$\begin{aligned} &A^{(k)} = B^{(k)} A = [\, I_{k|}, A^{(k)}_{|n-k} \,], \text{ and} \\ &B^{(k)} = J_k P_k [\, G_{k-1}, I_{|n-k+1} \,]\, \Pi_{k-1} = J_k [\, P_k G_{k-1}, I_{|n-k+1} \,]\, \Pi_k = [\, G_k, I_{|n-k} \,]\, \Pi_k, \end{aligned}$$

where $G_k = [J_k P_k G_{k-1}, g_k]$, and g_k is the Jordan vector. Note that $B^{(k)}$ is therefore the product of the first k scaled Jordan elimination matrices with pivoting, that is,

$$B^{(k)} = (J_k P_k) \cdots (J_2 P_2)(J_1 P_1).$$

We may call $B^{(k)}$ the aggregated (blocked) Jordan transformation. At the end of step n, $A^{(n)} = I_n = B^{(n)} A$, and finally $A^{-1} = B^{(n)} = G_n \Pi_n$.

For a fixed k, the permuted Jordan matrices, $P_i J_k P_i^{\mathrm{T}}$, $i > k$, have the same structure as J_k except that the Jordan vector is permuted. Each column of G_n represents a permuted Jordan matrix.

The merged algorithm for matrix inversion with partial pivoting (MAMIP) in Figure 2 is a one-sweep implementation of matrix inversion via GJE. It merges the storage for G_k and $A^{(k)}_{|n-k}$ and streamlines the computation sequence for successive transforms of both $A^{(k)}$ and $B^{(k)}$. The key feature of MAMIP lies in that it admits easy algorithm transforms to achieve better performance in different computing environments (see sections 5 and 7).

3 Jordan and Gauss

In this section we explain the mathematical and numerical connections between the approach using Gaussian elimination with partial pivoting and the approach using Jordan elimination with partial pivoting. We note first that

$$J_k = D_k^{-1}(I_n - v_k e_k^{\mathrm{T}}) = D_k^{-1} U_k^{-1} L_k^{-1} = D_k^{-1} L_k^{-1} U_k^{-1}, \quad k = 1 : n,$$

where

$$\begin{array}{l} U_k = (I_n + u_k e_k^{\mathrm{T}}), \quad L_k = (I_n + l_k e_k^{\mathrm{T}}), \quad v_k = u_k + l_k, \\ e_i^{\mathrm{T}} u_k = 0, \quad i \geq k, \quad e_i^{\mathrm{T}} l_k = 0, \quad i \leq k. \end{array}$$

That is, $(I_n - v_k e_k^{\mathrm{T}})$ is factored into two unit elementary triangular matrices, L_k and U_k. Notice that $L_n = U_1 = I_n$. Since $L_i D_j U_j = D_j U_j L_i$ for $i \geq j$, the following equation holds mathematically and numerically,

$$J_k \cdots J_2 J_1 = D_k^{-1} U_k^{-1} \cdots D_1^{-1} U_1^{-1} L_k^{-1} \cdots L_1^{-1}, \quad k = 1, 2, \cdots, n, \tag{4}$$

as long as the aggregation order is from the right to the left. This equivalent reordering reveals the implicit triangular factorization in the inverse method via GJE. Let $L^{(k)} = L_1 L_2 \cdots L_k$, and $U^{(k)} = U_1 D_1 U_2 D_2 \cdots U_k D_k$. Then, $L = L^{(n)}$ is unit lower triangular and $U = U^{(n)}$ is upper triangular. Computationally, the factors L and U are computed as in the matrix inversion method via Gaussian elimination though the factors are saved and inversed in a different way.

Specifically, let $A \leftarrow PA$, where P is the accumulated permutation matrix resulted from pivoting. Apply algorithm MAMIP to A, and consider $A^{(k)}$ the matrix generated after step k. Partition A as

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix},$$

where the leading square block A_{11} is of order k, and apply the same partition to $A^{(k)}$, $\hat{L}^{(k)} = (L^{(k)})^{-1}$, and $\hat{U}^{(k)} = (U^{(k)})^{-1}$. At step k of MAMIP not only L_k is generated as by Gaussian elimination, but the elementary factors U_k and D_k are also generated simultaneously (see the column scaling statement in MAMIP). As soon as generated, they are applied to the $n \times n$ array which now contains

$$A^{(k)} = \begin{pmatrix} \hat{U}_{11}^{(k)} \hat{L}_{11}^{(k)} & \hat{U}_{11}^{(k)} \hat{L}_{11}^{(k)} A_{11} \\ \hat{L}_{21}^{(k)} & \hat{L}_{11}^{(k)} A_{12} + A_{11} \end{pmatrix} = \begin{pmatrix} A_{11}^{-1} & A_{11}^{-1} A_{11} \\ A_{21} A_{11}^{-1} & A_{22} - A_{21} A_{11}^{-1} A_{12} \end{pmatrix}.$$

Note that the leading $k \times k$ block contains the inverse of A_{11} and the $(n-k) \times (n-k)$ trailing block contains $A_{22} - A_{21} A_{11}^{-1} A_{12}$, the Schur complement of A_{11} in A, as generated after k steps of Gaussian elimination (with partial pivoting).

4 Error Analysis

An error analysis for the inversion method via GJE is given in [12], treating the inversion as the solution of a system of linear equations with multiple right hand sides. We provide an error analysis in unification with the error analysis for other inversion methods that are explicitly or implicitly connected to the LU factorization and triangular solves.

We note that if $A^{(0)}$ is equal to a (unit) lower triangular matrix L, MAMIP is numerically equivalent to computing a right inverse of L by forward substitution, cf. Method 1 for inverting a triangular matrix in [12]. Similarly, if $A^{(0)} = U$ is (unit) upper triangular, MAMIP computes the left inverse of U by forward substitution, cf. [17]. Otherwise, in case $A^{(0)}$ is a full matrix, computing the inverse of $A^{(0)}$ can be verified to be equivalent to the following four steps (in comparison to the conventional inversion method as described in Section 1),

(1) LU factorization : $A^{(0)} = LU$,

(2) Triangular inversion : $LX = I_n$ (forward substitution),

(3) Triangular solve: $UY = X$ (forward substitution),

(4) $(A^{(0)})^{-1} = Y$.

This connection allows us to apply the error analysis for LU factorization and triangular solves. Since the identification of Step 2 and Step 3 is not presented explictly in the literature, we introduce an error analysis of the two steps, assuming L and U are given. We denote by $\tilde{B}$ the computed result of a mathematical quantity B. From the error analysis of a lower triangular solve by forward substitution, we have

$$(L + \Delta L_j)\tilde{X}e_j = e_j, \quad |\Delta L_j| \leq c_n u|L|, \quad j = 1 : n,$$

where u is the machine precision and c_n is a constant of $O(n)$, see [12, 21]. For an upper triangular solve by forward substitution, according to Peter and Wilkinson [17], we have

$$e_i^{\mathrm{T}}\tilde{Y}e_j = e_i^{\mathrm{T}}(U + \Delta U_{ij})^{-1}\tilde{X}e_j, \quad |\Delta U_{ij}| \leq c_n u|U|, \quad i = 1 : n, \quad j = 1 : n.$$

Let $R_x = L\tilde{X} - I_n$. Then, $\tilde{X} = L^{-1}(I_n + R_x)$ and $|R_x| \leq c_n u|L||\tilde{X}|$.

Many inversion methods have connections to LU factorization. An inverse process treats L (L^{-1}) and U (U^{-1}) as products of elementary triangular matrices. The difference lies in the computation ordering to aggregate the elementary triangular matrices. For example, the aggregation ordering of MAMIP is equivalent to the four steps described above.

A general approach for an error analysis of such an inversion method is (1) to identify the ordering of the triangular inverse (L or U, left inverse or right inverse) and the following triangular solve (left division or right division) and (2) to (provide and) apply the error analysis for the underlying triangular solve. If the triangular solve is based on forward or backward substitution, the composition of the absolute matrices in a forward error bound depends on the composition of the triangular factors in the computed inverse. Thus, many inversion methods have connections to LU factorization and can be analyzed in the same framework.

5 Variants of MAMIP

We describe in this section several variants of MAMIP for better performance in different computing environments.

```
% Input : A ∈ IR^{n×p}, n ≥ p, with full column rank.
% Output : Aggregated Jordan transformations overwritten onto A.
ipiv = [1 : n];
for k = 1 : p
    % updating the pivot column
    A(:, k) = A(ipiv, k);
    A(:, k) = A(:, 1:k−1) ∗ A(1:k−1, k) + [ zeros(k−1, 1); A(k:n, k) ];
    % pivoting
    [ abspivot, k̄ ] = max( [ zeros(k−1, 1); abs(A(k:n, k)) ] );
    pivot = A(k̄, k);
    [ A(k, 1:k), A(k̄, 1:k)] = swap(A(k, 1:k), A(k̄, 1:k));
    [ ipiv(k), ipiv(k̄) ] = swap( ipiv(k), ipiv(k̄) );
    % Jordan transformation
    A(:, k)         = −[ A(1:k−1, k); 0; A(k+1:n, k) ] / pivot;
    A(:, 1:k−1)  = A(:, 1:k−1) + A(:, k) ∗ A(k, 1:k−1);
    A(k, 1:k)      = [ A(k, 1:k−1), 1 ] / pivot;
end
A(:, ipiv) = A;        % backward permutation
```

Figure 3: Incremental version of MAMIP

5.1 An Incremental Inverse Algorithm

Recall that at the end of step $k-1$ of MAMIP, $1 < k \leq n$, we have

$$[\, G_{k-1} \,|\, A^{(k-1)}_{|n-k+1} \,] \quad \text{and} \quad \Pi_{k-1},$$

where $(G_{k-1}, I_{|n-k+1})\Pi_{k-1} = B^{(k-1)}$ is the product of the first $k-1$ scaled Jordan transformations with partial pivoting, and $A^{(k-1)}_{|n-k+1}$ consists of the last $n-k+1$ columns of the original matrix A transformed by $B^{(k-1)}$. In the algorithm listed in Figure 3 the k-th column of the original matrix is not updated until step k, $1 \leq k \leq n$. The same idea is previously applied in the left-looking version of the LU factorization [9].

```
% Input : A ∈ IR^{n×n} nonsingular and nb the block size.
% Output : A^{-1} ∈ IR^{n×n} overwritten onto A.
ipiv = [1 : n];
for k = 1 : nb : n
    kb = min(nb, n − k + 1);
    in = [k : k + kb − 1];  out = [1:k−1, k+kb:n];
    [ A(:, in), ibpiv ] = MAMIP_INC2(k, A(:, in));
    A(:, out) = A(ibpiv, out);
    A(out, out) = A(out, out) + A(out, in) ∗ A(in, out);
    A(in, out) = A(in, in) ∗ A(in, out);
    ipiv = ipiv(ibpiv);
end
A(:, ipiv) = A;            % backward permutation
```

Figure 4: A blocked variant of MAMIP

5.2 Blocked Algorithms

We note that the incremental algorithm can be applied to a subset of the columns. In fact, we use it in the blocked algorithm listed in Figure 4 to construct a blockwise Gauss-Jordan transform. For each block of kb columns, we use the incremental algorithm to form the product of kb Jordan transformations with partial pivoting in the compact form of kb columns and a permutation matrix, which is represented by a vector ibpiv. The subroutine MAMIP_INC2 is based on the incremental algorithm with an offset, k, to the indexing in the partial pivoting. The rest of the matrix is then permuted in rows accordingly and updated by the aggregated Jordan transformation in Level-3 BLAS operations. As for the algorithmic block size, it is desirable to choose the block size big enough to reduce data migration between hierarchical memory levels in uni-processors. In multiprocessors with distributed memory, the distribution block size plays an analogous role. On the other hand, the block size should be kept small enough to keep the numerical accuracy close to that of the non-blocked algorithm. The tradeoff or compromise between speed performance and numerical properties varies with applications requirements.

5.3 Parallel Implementation Issues

The blocked algorithm can be implemented in different ways for parallel computation, depending on the underlying system(s) and the programming environment. We discuss in the following several common issues on parallel implementation and performance optimization on parallel computers with distributed memory. The advantages of MAMIP are due to the fact that MAMIP circumvents the use of the intermediate triangular matrices from LU factorizations.

- MAMIP admits a non-square mesh of nodes without complicating the programming. This renders an additional freedom in mesh configuration for performance enhancement.
- MAMIP allows the decoupling between algorithmic block size and the physical panel size for distributed data layout. Such property makes MAMIP free of the constraints on data alignment or panel size posed by a software library for parallel computing.
- MAMIP can be easily implemented with variant algorithmic block size and variant panel size, for example, for better job balancing on a cluster of heterogeneous multiple processors.
- The communication pattern for MAMIP is simple. In a message-passing programming environment, the communication channels and the message lengths (except for partial pivoting) can be set up once for all, since they remain the same throughout the algorithm.

6 Performance model

To study the theoretical performance of our parallel algorithm, we consider an $n \times n$ matrix A distributed among a $p = r \times c$ grid of nodes, using a 2-D block scattered (BS) decomposition (see, e.g., [6]) with square block size equal to $n_b \times n_b$. For the sake of simplicity, we also assume the following:

- The time of a floating-point arithmetic operation is τ.
- The time time of transfering m floating-point numbers between any two nodes is $\alpha + \beta m$. Here, α is the communication latency and β is the inverse of the bandwith ($\alpha \gg \beta$).

- The time of transfering a message of m floating-point numbers to Q nodes (broadcast) is $\log(Q)(\alpha+\beta m)$, with $\log(Q)=\log_2(Q)$.
- r, c, and n_b divide n.

Our parallel algorithm is based on a block-partitioned version of MAMIP. In this blocked version, matrix A is processed by blocks of n_b columns. At stage k, first the k-th column block of A is processed by means of MAMIP_INC2; then, the row permutations required by k-th column block are further applied to blocks $1:k-1$ and $k+1:n/n_b$, and these blocks are processed by means of two rank-n_b updates. The approximate time of the stages in iteration k are the following (the lower order terms are neglected):

1. Apply MAMIP_INC2 to the columns of the current column block, $A(:,k:k+n_b-1)$:
   (a) Determine the pivot row: $T_1=\alpha\log(r)$.
   (b) Swap the pivot row and the current row: $T_2=\alpha+\beta\, n_b$.
   (c) Broadcast the pivot row to the rest of nodes: $T_3=(\alpha+\beta\, n_b)\log(r)$
   (d) Update the local column block: $T_4=2\tau\, n_b\, n/r$.
2. Apply transformations to the rest of the matrix.
   (a) Broadcast pivot information of k-th column block: $T_5=(\alpha+\beta\, n_b)\log(c)$.
   (b) Swap the corresponding rows: $T_6=\alpha+(n-n_b)/c\beta$.
   (c) Broadcast information for the rank-n_b update: $T_7=(\alpha+n/r\, n_b\,\beta)\log(c)$ (broadcast k-th block) and $T_8=(\alpha+(n-n_b)/c\, n_b\,\beta)\log(r)$ (broadcast k-th block row).
   (d) Rank-n_b update of the rest of the matrix: $T_9=2\tau\, n_b\, n(n-n_b)/p$.

The previous procedure is repeated for $k=1{:}n/n_b$. If no overlapping between communication and computation is assumed we obtain the following time

$$C_{r\times c}=(T_1+T_2+\cdots+T_4)n+(T_5+T_6+\cdots+T_9)n/n_b,$$

which is an upper bound of the global time of the parallel algorithm. The *speed-up*, or acceleration of the algorithm in $r\times c$ nodes, is then

$$S_{r\times c}=2\tau n^3/C_{r\times c},$$

which shows the asymptotic convergence of $S_{r\times c}$ to p.

7 Experimental Results

We present results for three implementations of matrix inversion:

`SL_IGEP` is part of ScaLAPACK [6] and implements the "traditional" inversion algorithm via Gaussian elimination;

`SL_IGJEP` was coded using ScaLAPACK parallel BLAS kernels (PBLAS). It implements the parallel algorithm analysed in the previous section. This implementation assumes that the algorithmic block size equals the distribution block size, that all operations required to invert the current diagonal block are performed on a single node, and that the current column block exists and is updated within a single column of nodes; and

`PLA_IGJEP` was coded using PLAPACK [20]. This code has some additional features: The current column block is redistributed so that all nodes participate when MAMIP_INC2 is applied. This allows for better load balance during this stage of the algorithm. It also simplifies the coding when the algorithmic block size is to be larger than the distribution block size and simplifies the addition of multiple levels of blocking to MAMIP_INC2 while it is being applied to the current column block.

Performance results are given for the Cray T3E-600 (300 MHz), with all computations performed in 64-bit arithmetic. Version information is given as follows:

UNICOS/mk	2.0.3.10
C compiler	6.0.0.0
F90 compiler	3.0.2.0
Assembler	2.3.0.0
Cray Libs	3.0.0.0
CrayTools	3.0.0.0
Options	-O3
Streams	on

We used ScaLAPACK available from netlib (version 1.5 + update (version 1.6)), with the MPIBLACS compiled for Cray T3E using Cray's MPI. The PLAPACK version, to be released shortly as Version R.1.2, also uses MPI. We report performance measuring MFLOPS/sec. (millions of floating point operations per second) using an operation count of $2n^3$ for matrix inversion for all versions.

In Fig. 5, we report performance as a function of the matrix size for various numbers of nodes. Very respectable performance is attained by all implementations: the highly optimized massively parallel LINPACK benchmark on the Cray T3E-600 attains around 440 MFLOPS/node on 32 nodes, for a much larger problem (21120×21120) than we used in our measurements. In these measurements, algorithmic and distribution block size of 48 was used for both `SL_IGEP` and `SL_IGJEP`. As expected, `SL_IGJEP` performs better than `SL_IGEP` since it incurs less communication and exhibits better load balance. For the more flexible implementation `PLA_IGJEP` we used a distribution block size of 24 and an algorithmic block size of 96. Since an algorithmic block size of 96 allows local matrix-matrix multiplication to attain higher performance, asymptotically `PLA_IGJEP` attains better performance than either `SL_IGEP` or `SL_IGJEP`. For smaller problems, the additional cost of redistributing the current column block results in a modest reduction in performance.

In Fig. 6(a)–(c) we report performance as a function of the number of nodes, keeping local memory utilization constant. In this figure, the label `local NxN` indicates that local memory use is equal to that required to locally store an `NxN` matrix. Notice that as more nodes are employed, performance per node remains approximately constant, indicating excellent scalability.

In Fig. 6(d), we demonstrate how performance is affected by the distribution block size, n_b. For the `SL_IGEP` and `SL_IGJEP`, smaller n_b means better load balance, but also slower performance of the local matrix-matrix multiplication. Thus, $n_b = 48$ turns out to be a good compromise. For `PLA_IGJEP`, the distribution block size does not dictate the algorithmic block size, and thus a smaller n_b does not affect the performance of the local matrix-matrix multiplication. Thus, for this implementation, a smaller distribution block size yields better load-balance and better overall performance.

We conclude by indicating that for the ScaLAPACK based implementations we could have instead used the ScaLAPACK version provided by the Cray Scientific library. This version uses the *shmem* library, which greatly reduces communication overhead. In order to provide a fair comparison between `PLA_IGJEP` and `SL_IGJEP`, we chose to use the MPI based version of ScaLAPACK instead. Asymptotically, the performance of either ScaLAPACK version is similar, but for small matrices, there is a definitive advantage to using the *shmem* based version.

8 Concluding Remarks

We have related the matrix inversion via Gauss-Jordan elimination with the traditional matrix inversion by means of Gaussian elimination. The Gauss-Jordan approach

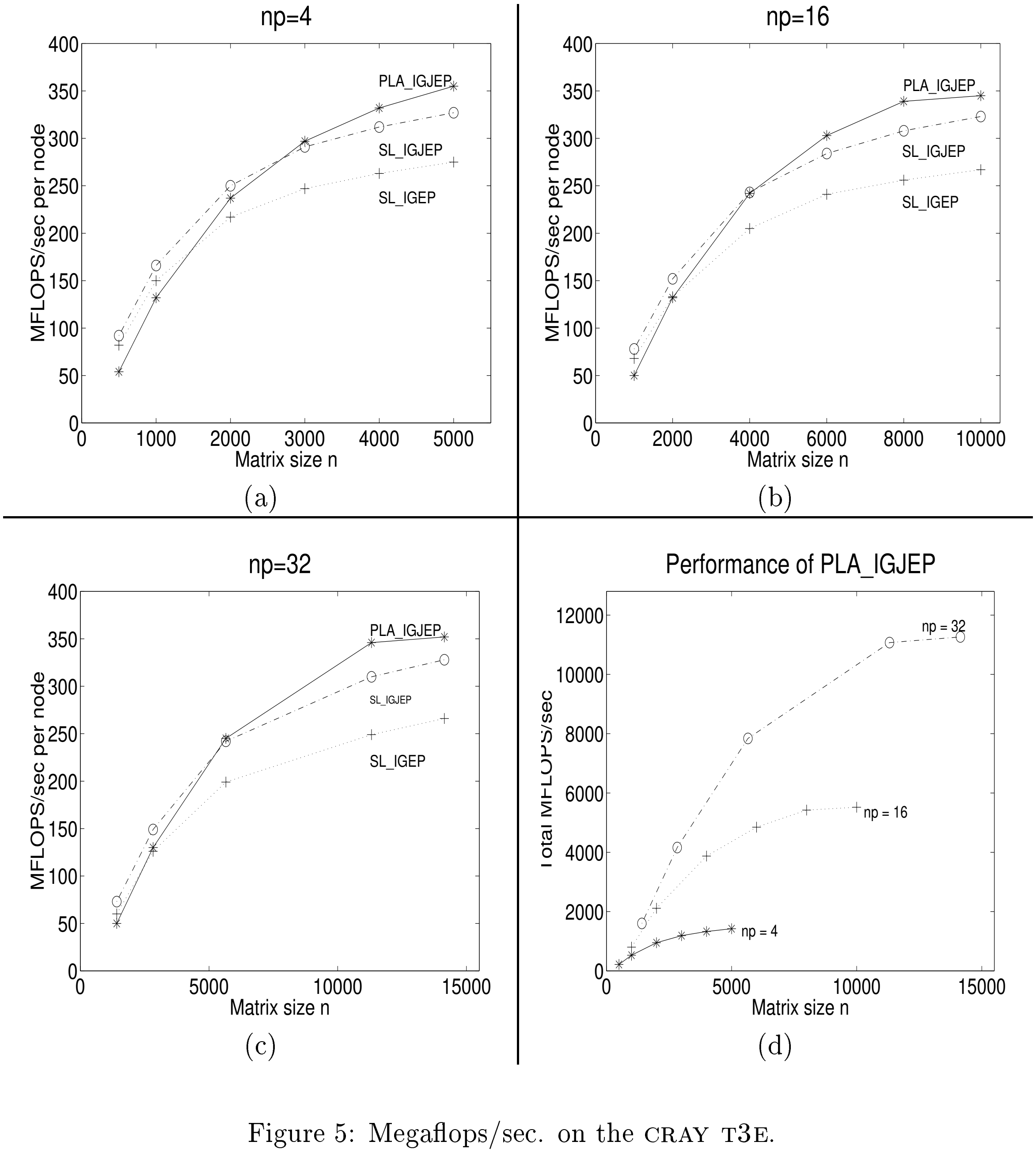

Figure 5: Megaflops/sec. on the CRAY T3E.

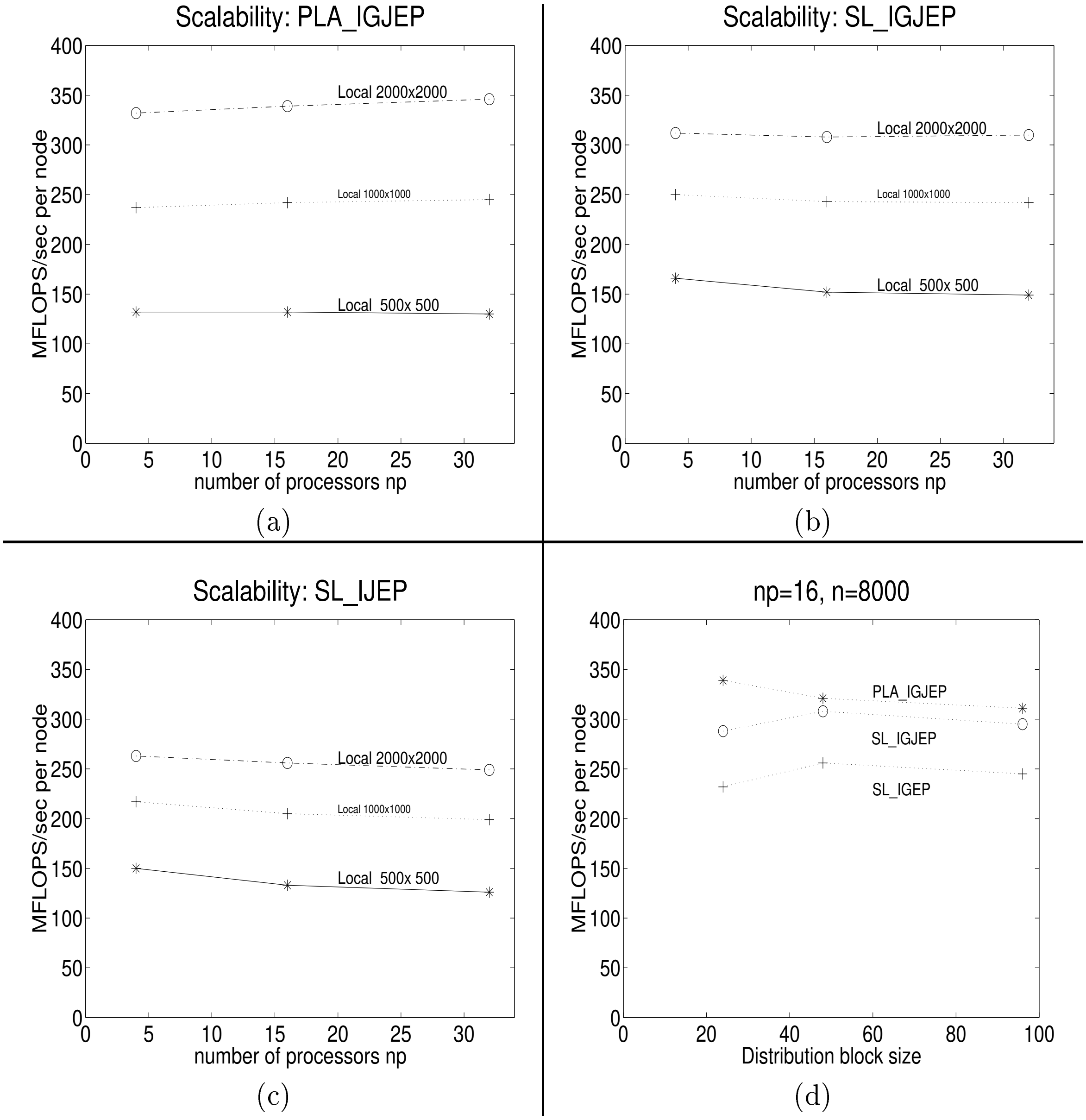

Figure 6: Scalability and performance vs. distribution block size on the CRAY T3E.

presents the same arithmetic cost as the conventional inversion algorithm while maintaining its numerical properties. Our matrix inversion algorithm renders natural load balance, simple programming, and easy performance optimization, which are specially appropriate for parallel computers with distributed memory. The experimental results on an CRAY T3E report the high performance and scalability of the new parallel matrix inversion algorithm.

Implementations for `SL_IGJEP` and `PLA_IGJEP` are available in `http://www.cs.utexas.edu/users/plapack/inverse`.

Acknowledgments

We express our gratitude to Y. J. Wu for his help and his valuable comments and suggestions.

Primary support for this project came from the Parallel Research on Invariant Subspace Methods (PRISM) project (ARPA grant P-95006) and a grant from the Intel Research Council.

Access to equipment was provided by the National Partnership for Advanced Computational Infrastructure (NPACI) and The University of Texas Advanced Computing Center (TACC).

E. S. Quintana and G. Quintana received support from the CICYT project TIC 96-1062-C03-03. E. S. Quintana was also supported by the *Consellería de Cultura Educación y Ciencia* of the *Generalitat Valenciana.*

References

[1] 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.

[2] Z. Bai, J. Demmel, J. Dongarra, A. Petitet, H. Robinson, and K. Stanley. The spectral decomposition of nonsymmetric matrices on distributed memory parallel computers. *SIAM J. on Scientific Computing*, 18:1446–1461, 1997.

[3] G. Baker. *Implementation of parallel processing to selected problems in satellite geodesy.* PhD thesis, Department of Aerospace Engineering, The University of Texas at Austin, 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. *Journal 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. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. van der Vost. *Solving linear systems on vector and shared memory computers*. SIAM, 1990.

[10] P. Geng, J.T. Oden, and R. van de Geijn. Massively parallel computation for acoustical scattering problems using boundary element methods. *Journal of Sound and Vibration*, 191(1):145–165, 1997.

[11] 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. 26 pp.

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

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

[14] A.J. Laub and J.D. Gardiner. Hypercube implementation of some parallel algorithms in control. In M.J. Denham and A.J. Laub, editors, *Advanced Computing Concepts and Techniques in Control Engineering*, pages 361–390. Springer-Verlag, Berlin, 1988.

[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] G. Peters and J. H. Wilkinson. On the stability of Gauss-Jordan elimination with pivoting. *Comm. of the ACM*, 18(1):20–24, 1975.

[18] 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.

[19] G. W. Stewart. *Introduction to matrix computations.* Academic Press, New York, NY, 1973.

[20] R. van de Geijn. *Using PLAPACK.* The MIT Press, 1997.

[21] J. H. Wilkinson. *Rounding errors in algebraic processes.* Prentice-Hall, Englewood Cliffs, NJ, 1963.