PLAPACK: High Performance through High Level Abstraction

Greg Baker* John Gunnels Greg Morrow Beatrice Riviere
Robert van de Geijn†

The University of Texas at Austin
Austin, TX 78712

A Technical Paper Submitted to ICPP'98
February 10, 1998

Abstract

Coding parallel algorithms is generally regarded as a formidable task. To make this task manageable in the arena of linear algebra algorithms, we have developed the Parallel Linear Algebra Package (PLAPACK), an infrastructure for coding such algorithms at a high level of abstraction. It is often believed that by raising the level of abstraction in this fashion, performance is sacrificed. Throughout, we have maintained that indeed there is a performance penalty, but that by coding at a higher level of abstraction, more sophisticated algorithms can be implemented, which allows high levels of performance to be regained. In this paper, we show this to indeed be the case for the parallel solver package implemented using PLAPACK, which includes Cholesky, LU, and QR factorization based solvers for symmetric positive definite, general, and overdetermined systems of equations, respectively. Performance comparison with ScaLAPACK shows *better* performance is attained by our solvers.

1 Introduction

In this paper, we show that by creating an infrastructure for high level specification of parallel dense linear algebra algorithms, not only is the code greatly simplified, but higher performance is achieved. Due to the simplicity of the specification of the code, more complex implementation can be attempted, which in turn yield higher performance. We demonstrate the validity of his observation through the use of the Parallel Linear Algebra Package (PLAPACK) infrastructure to code high performance parallel implementations of matrix decomposition algorithms like those for the Cholesky, LU, and QR factorizations. What is somewhat surprising is that despite the fact that the literature on implementation of these algorithms is vast (indeed too large to properly cite in this paper), we derive new algorithms for these factorizations, which are shown to yield superior performance.

High performance libraries for matrix operations like the various factorizations first became available with the arrival of the LAPACK [2] library in the early 90's. This library recognized the necessity to code in terms of Basic Linear Algebra Subprograms (BLAS) [12, 7, 6] for modularity and portable performance. In particular, it explores the benefits of recoding such algorithms in terms of matrix-matrix kernels like matrix-matrix multiplication in order to improve utilization of the cache of a microprocessor.

*Currently with Lockheed Martin Federal Systems, Inc., 700 N. Frederick Avenue, Gaithersburg, MD 20879-3328
†Corresponding Author. Department of Computer Sciences, The University of Texas, Austin, TX 78712, (512) 471-9720, (512) 471-8885 (fax), rvdg@cs.utexas.edu

Extensions of LAPACK to distributed memory architectures like the Cray T3E, IBM SP-2, and Intel Paragon are explored by the ScaLAPACK project [4]. While this package does manage to provide a subset of the functionality of LAPACK, it is our belief that it has also clearly demonstrated that mimicking the coding styles that were effective on sequential and shared memory computers does not create maintainable and flexible code for distributed memory architectures. The PLAPACK infrastructure attempts to show that by adopting an object based coding style, already popularized by the Message-Passing Infrastructure (MPI) [10, 14], the coding of *parallel* linear algebra algorithms is *simplified* compared to the more traditional *sequential* coding approaches. We contrast the coding styles in [1].

It is generally believed that by coding at a higher level of abstraction, one pays a price of higher overhead and thus reduced performance. While it is certainly true that the PLAPACK infrastructure does impose such an overhead, we demonstrate that one can overcome this overhead by implementing more complex algorithms. One can argue that these more complex algorithms can always also be implemented using more traditional methods. However, it is the higher level of abstraction that makes the task manageable, especially when a large number of algorithms are to be implemented.

This paper is structured as follows: In Section 2, we explain matrix-vector and matrix-matrix operation based versions of an algorithm for computing the Cholesky factorization of a symmetric positive definite matrix. We show how this algorithm is parallelized and coded using PLAPACK, progressively incorporating more sophistication into the implementation. The final improvement yields a previously unpublished algorithm, which is shown to outperform previously published implementations. In Sections 3–4, we similarly discuss parallel implementation of the LU and QR factorizations. In Section 5, we present initial performance results on the Cray T3E, including a comparison with ScaLAPACK. Concluding remarks are given in the final section.

2 Cholesky Factorization

In this section, we discuss the computation of the Cholesky factorization

$$A = LL^T$$

where A is an $n \times n$ symmetric positive definite matrix and L is an $n \times n$ lowertriangular matrix.

2.1 Basic algorithm

The right-looking algorithm for implementing this operation can be described by partitioning the matrices

$$A = \left(\begin{array}{c|c} \alpha_{11} & \star \\ \hline a_{21} & A_{22} \end{array} \right) \quad \text{and} \quad L = \left(\begin{array}{c|c} \lambda_{11} & 0 \\ \hline l_{21} & L_{22} \end{array} \right)$$

where α_{11} and λ_{11} are scalars. As a result, a_{21} and l_{21} are vectors of length $n-1$, and A_{22} and L_{22} are matrices of size $(n-1) \times (n-1)$. The $\star$ indicates the symmetric part of A, which will not be updated. Now,

$$A = \left(\begin{array}{c|c} a_{11} & \star \\ \hline a_{21} & A_{22} \end{array} \right) = \left(\begin{array}{c|c} \lambda_{11} & 0 \\ \hline l_{21} & L_{22} \end{array} \right) \left(\begin{array}{c|c} \lambda_{11} & l_{21}^T \\ \hline 0 & L_{22}^T \end{array} \right) = \left(\begin{array}{c|c} \lambda_{11}^2 & \star \\ \hline \lambda_{11} l_{21} & l_{21} l_{21}^T + L_{22} L_{22}^T \end{array} \right)$$

From this we derive the equations

$$\begin{array}{rcl} \lambda_{11} & = & \sqrt{\alpha_{11}} \\ l_{21} & = & a_{21} / \lambda_{11} \\ A_{22} - l_{21} l_{21}^T & = & L_{22} L_{22}^T \end{array}$$

An algorithm for computing the Cholesky factorization is now given by

1. Partition $A = \left(\begin{array}{c|c} \alpha_{11} & \star \\ \hline a_{21} & A_{22} \end{array}\right)$
2. $\alpha_{11} \leftarrow \lambda_{11} = \sqrt{\alpha_{11}}$
3. $a_{21} \leftarrow l_{21} = a_{21}/\lambda_{11}$
4. $A_{22} \leftarrow A_{22} - l_{21} l_{21}^T$
5. continue recursively with A_{22}

Note that only the upper or lower triangular part of a symmetric matrix needs to be stored, and the above algorithm only updates the lower portion of the matrix with the result L. As a result, in the step $A_{22} \leftarrow A_{22} - l_{21} l_{21}^T$ only the lower portion of A_{22} is updated, which is typically referred to as a *symmetric rank-1 update*.

One question that may be asked about the above algorithm is what is stored in the matrix after k steps. We answer this by partitioning

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

where A_{TL} and L_{TL} are $k \times k$. Here "TL" , "BL", and "BR" stand for "Top-Left", "Bottom-Left", and "Bottom-Right", respectively. As seen before

$$A = \left(\begin{array}{c|c} A_{TL} & \star \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \left(\begin{array}{c|c} L_{TL}^T & L_{BL}^T \\ \hline 0 & L_{BR}^T \end{array}\right) = \left(\begin{array}{c|c} L_{TL} L_{TL}^T & \star \\ \hline L_{BL} L_{TL}^T & L_{BL} L_{BL}^T + L_{BR} L_{BR}^T \end{array}\right)$$

so that

$$\begin{array}{rcl} A_{TL} & = & L_{TL} L_{TL}^T \\ A_{BL} & = & L_{BL} L_{TL}^T \\ A_{BR} & = & L_{BR} L_{BR}^T + L_{BL} L_{BL}^T \end{array}$$

It can be easily verified that the above algorithm has the effect that after k steps

- A_{TL} has been overwritten by L_{TL},
- A_{BL} has been overwritten by L_{BL}, and
- A_{BR} has been overwritten by $A_{BR} - L_{BL} L_{BL}^T$.

Thus, the matrix with which the algorithm is continued at each step is the submatrix $A_{BR}$and to complete the Cholesky factorization, it suffices to compute the factorization of the updated A_{BR}. This motivates the algorithm given on the left in Fig. 1.

2.2 Blocked algorithm

The bulk of the computation in the above algorithm is in the symmetric rank-1 update, which performs $O(n^2)$ operations on $O(n^2)$ data. It is this ratio of computation to data volume (requiring memory accesses) that stands in the way of high performance. To overcome this, we now show how to reformulate the algorithm in terms of matrix-matrix multiplication (or rank-k updates) which have a more favorable computation to data volume ratio, allowing for more effective use of a microprocessor's cache memory.

Given the derivation of the basic algorithm given above, it is not difficult to derive a blocked (matrix-matrix operation based) algorithm, as is shown in Fig. 1.

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

Assume A_{TL} and A_{BL} have been overwritten by L_{TL} and L_{BL} respectively, and A_{BR} has been overwritten by $A_{BR}^{\text{new}} = A_{BR} - L_{BL}L_{BL}^T$ so that only the Cholesky factorization of A_{BR}^{new} needs to be computed.

$$A_{BR}^{\text{new}} = \left(\begin{array}{c|c} \alpha_{11} & \star \\ \hline a_{21} & A_{22} \end{array}\right) \quad \text{and} \quad L_{BR} = \left(\begin{array}{c|c} \lambda_{11} & 0 \\ \hline l_{21} & L_{22} \end{array}\right)$$

where α_{11} and λ_{11} are scalars. Then

$$\begin{array}{rcl} A_{BR}^{\text{new}} & = & \left(\begin{array}{c|c} \alpha_{11} & \star \\ \hline a_{21} & A_{22} \end{array}\right) \\ & = & \left(\begin{array}{c|c} \lambda_{11}^2 & \star \\ \hline \lambda_{11}l_{21} & l_{21}l_{21}^T + L_{22}L_{22}^T \end{array}\right) = L_{BR}L_{BR}^T \end{array}$$

From this we derive the equations

$$\begin{array}{rcl} \lambda_{11} & = & \sqrt{\alpha_{11}} \\ l_{21} & = & a_{21}/\lambda_{11} \\ A_{22} - l_{21}l_{21}^T & = & L_{22}L_{22}^T \end{array}$$

The above motivation yields the algorithm

partition $A = \left(\begin{array}{c|c} A_{TL} & \star \\ \hline A_{BL} & A_{BR} \end{array}\right)$
where A_{TL} is 0×0
do until A_{BR} is 0×0

repartition

$$\left(\begin{array}{c||c} A_{TL} & \star \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c||c|c} A_{00} & \star & \star \\ \hline\hline a_{10}^T & \alpha_{11} & \star \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right)$$

$\alpha_{11} \leftarrow \lambda_{11} = \sqrt{\alpha_{11}}$

$a_{21} \leftarrow l_{21} = a_{21}/\lambda_{11}$

$A_{22} \leftarrow A_{22} - l_{21}l_{21}^T$

continue with

$$\left(\begin{array}{c||c} A_{TL} & \star \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c|c||c} A_{00} & \star & \star \\ \hline a_{10}^T & \alpha_{11} & \star \\ \hline\hline A_{20} & a_{21} & A_{22} \end{array}\right)$$

enddo

$$A_{BR}^{\text{new}} = \left(\begin{array}{c|c} A_{11} & \star \\ \hline A_{21} & A_{22} \end{array}\right) \quad \text{and} \quad L_{BR} = \left(\begin{array}{c|c} L_{11} & 0 \\ \hline L_{21} & L_{22} \end{array}\right)$$

where A_{11} and L_{11} are of size $b \times b$. Then

$$\begin{array}{rcl} A_{BR}^{\text{new}} & = & \left(\begin{array}{c|c} A_{11} & \star \\ \hline A_{21} & A_{22} \end{array}\right) \\ & = & \left(\begin{array}{c|c} L_{11}L_{11}^T & \star \\ \hline L_{21}L_{11}^T & L_{21}L_{21}^T + L_{22}L_{22}^T \end{array}\right) = L_{BR}L_{BR}^T \end{array}$$

From this we derive the equations

$$\begin{array}{rcl} A_{11} & = & L_{11}L_{11}^T \\ L_{21}L_{11}^T & = & A_{21} \\ A_{22} - L_{21}L_{21}^T & = & L_{22}L_{22}^T \end{array}$$

The above motivation yields the algorithm

partition $A = \left(\begin{array}{c|c} A_{TL} & \star \\ \hline A_{BL} & A_{BR} \end{array}\right)$
where A_{TL} is 0×0
do until A_{BR} is 0×0
determine block size b
repartition

$$\left(\begin{array}{c||c} A_{TL} & \star \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c||c|c} A_{00} & \star & \star \\ \hline\hline A_{10} & A_{11} & \star \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

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

$A_{11} \leftarrow L_{11} = \text{Chol. fact.}(A_{11})$

$A_{21} \leftarrow L_{21} = A_{21}L_{11}^{-T}$

$A_{22} \leftarrow A_{22} - L_{21}L_{21}^T$

continue with

$$\left(\begin{array}{c||c} A_{TL} & \star \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c|c||c} A_{00} & \star & \star \\ \hline A_{10} & A_{11} & \star \\ \hline\hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

enddo

Figure 1: Similarity in the derivation of a Level-2 (*left*) and Level-3 (*right*) BLAS based right-looking Cholesky factorization.

2.3 PLAPACK implementation

In PLAPACK, information (e.g. size and distribution) regarding a linear algebra object (e.g. matrix or vector) is encoded in a data structure (opaque object) much like MPI encodes communicators. Thus, the calling the calling sequence for a Cholesky factorization need only have one parameter, the object that describes the matrix. One advantage of this approach is that references into the same data can be created as new objects, called *views*. PLAPACK provides routines that query information associated with an object and other routines that create views. Finally, parallelizing the above blocked algorithm in essense comes down to parallelizing the different major operations:

- Cholesky factorization of A_{11} $\quad A_{11} \leftarrow L_{11} = \text{Chol. fact}(A_{11})$
- Update of A_{21} $\quad A_{21} \leftarrow L_{21} = A_{21} L_{11}^{-T}$ (solved as triangular solve with multiple-right-hand-sides $L_{11} L_{21}^T = A_{21}^T$)
- Symmetric rank-k update of A_{22} $\quad A_{22} \leftarrow A_{22} - L_{21} L_{21}^T$

PLAPACK provides parallel kernels that perform these operations.

Basic parallel implementation

Fig. 2 (a)–(b) illustrate how the developed algorithm is translated into a PLAPACK code. It suffices to know that object `A` references the original matrix to be factored. Object `ABR` is a view into the current part of the matrix that still needs to be factored (A_{BR} in previous discussion). The remainder of the code is self explanatory when compared to the algorithm in (a) of that figure. Note that the call to `Chol_level2` is a call to a basic (nonblocked) implementation of the Cholesky factorization. That routine itself resembles the presented code in (b) closely.

Basic optimizations

While the code presented in Fig. 2 (b) is a straight forward translation of the developed algorithm, it does not provide high performance. The primary reason for this is that the factorization of A_{11} is being performed as a call to a parallel matrix-vector based routine, which requires an extremely large number of communications. These communications can be avoided by choosing the size of A_{11} so that it exists entirely on one node.

In Fig. 2 (c) we show how minor modifications to the PLAPACK implementation in (b) allows us to force A_{11} to exist on one node. To understand the optimizations, one must have a rudimentary understanding of how matrices are distributed by PLAPACK. A given matrix A is partitioned like

$$A = \left(\begin{array}{c|c|c} A_{0,0} & \cdots & A_{0,(N-1)} \\ \hline \vdots & & \vdots \\ \hline A_{(M-1),0} & \cdots & A_{(M-1),(N-1)} \end{array} \right)$$

For understanding the code, the sizes of the blocks are not important. What *is* important is that these blocks are assigned to an $r \times c$ logical mesh of nodes using a two-dimensional cartesian distribution:

- All blocks in the same row of blocks, $A_{i,*}$ are assigned to the same row of nodes.
- All blocks in the same column of blocks, $A_{*,j}$ are assigned to the same column of nodes.

.

Given that the currently active submatrix A_{BR} is distributed to a given logical mesh of nodes, determining a block size so that A_{11} resides on one node requires us to take the minimum of

- the number of columns that can be split off from the left of A_{BR} while remaining in the same block of columns of A_{BR}, and
- the number of rows that can be split off from the top of A_{BR} while remaining in the same block of rows of A_{BR}.

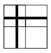
partition $A = \left(\begin{array}{c|c} A_{TL} & \star \\ \hline A_{BL} & A_{BR} \end{array} \right)$

where A_{TL} is 0×0

do until A_{BR} is 0×0

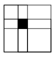
determine block size b

repartition

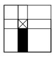


$$\left(\begin{array}{c||c} A_{TL} & \star \\ \hline\hline A_{BL} & A_{BR} \end{array} \right) = \left(\begin{array}{c||c|c} A_{00} & \star & \star \\ \hline\hline A_{10} & A_{11} & \star \\ \hline A_{20} & A_{21} & A_{22} \end{array} \right)$$

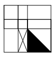
where A_{11} is $b \times b$



$A_{11} \leftarrow L_{11} = \text{Chol. fact.}(A_{11})$

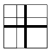


$A_{21} \leftarrow L_{21} = A_{21} L_{11}^{-T}$



$A_{22} \leftarrow A_{22} - L_{21} L_{21}^{T}$

continue with



$$\left(\begin{array}{c||c} A_{TL} & \star \\ \hline\hline A_{BL} & A_{BR} \end{array} \right) = \left(\begin{array}{c|c||c} A_{00} & \star & \star \\ \hline A_{10} & A_{11} & \star \\ \hline\hline A_{20} & A_{21} & A_{22} \end{array} \right)$$

enddo

(a) Blocked algorithm

```
PLA_Obj_view_all( A, &ABR );
while ( TRUE ) {
  PLA_Obj_global_length( ABR , &b );
  b = min( b, nb );
  if ( 0 == b ) break;

  PLA_Obj_split_4( ABR, b, b, &A11, &A12,
                              &A21, &ABR );

  Chol_level2( A11 );

  PLA_Trsm( PLA_SIDE_RIGHT, PLA_LOW_TRIAN,
            PLA_TRANS, PLA_NONUNIT_DIAG,
            one, A11, A21 );
  PLA_Syrk( PLA_LOW_TRIAN, PLA_NO_TRANS,
                minus_one, A21, one, ABR );
}
```

(b) Basic PLAPACK implementation

```
PLA_Obj_view_all( A, &ABR );
while ( TRUE ) {
  PLA_Obj_split_size( ABR, PLA_SIDE_LEFT,
                      &b_l, &owner_l );
  PLA_Obj_split_size( ABR, PLA_SIDE_TOP,
                      &b_t, &owner_t );
  b = min( min( b_t, b_l ), nb );
  if ( 0 == b ) break;

  PLA_Obj_split_4( ABR, b, b, &A11, &A12,
                              &A21, &ABR );
  PLA_Obj_objtype_cast( A11, PLA_MSCALAR );

  PLA_Local_chol( A11 );

  PLA_Mscalar_create_conf_to( A11,
        PLA_ALL_ROWS, PLA_INHERIT, &A11_dup );
  PLA_Copy( A11, A11_dup );

  PLA_Local_trsm( PLA_SIDE_RIGHT, PLA_LOW_TRIAN,
                  PLA_TRANS, PLA_NONUNIT_DIAG,
                  one, A11_dup, A21 );
  PLA_Syrk( PLA_LOW_TRIAN, PLA_NO_TRANS,
            minus_one, A21, one, ABR );
}
```

(c) Basic optimizations

```
PLA_Obj_split_4( A, 0, 0, &ATL, &ATR,
                          &ABL, &ABR );
PLA_Mvector_create_conf_to( ABL, nb, &A1_mv );

while ( TRUE ) {
  PLA_Obj_global_length( ABR, &b );
  b = min( b, nb );
  if ( 0 == b ) break;

  PLA_Obj_vert_split_2( ABR,   b, &A1,     &A2 );
  PLA_Obj_vert_split_2( A1_mv, b, &A1_mv1, &A1_mv2 );

  PLA_Copy( A1, A1_mv1 );
  PLA_Chol_mv( A1_mv1 );
  PLA_Copy( A1_mv1, A1 );

  PLA_Obj_split_4( ABR, b, b, &A11, &A12,
                              &A21, &ABR );
  PLA_Syrk( PLA_LOW_TRIAN, PLA_NO_TRANS,
            minus_one, A21, one, ABR );

  PLA_Obj_horz_split_2( A1_mv, b, PLA_DUMMY,
                                  &A1_mv );
}
```

(d) New parallel algorithm

Figure 2: Various versions of level-3 BLAS based Cholesky factorization using PLAPACK.

Once it is guaranteed that A_{11} resides within one node, the call to `Chol_level2` can be replaced by a sequential factorization provided by `PLA_Local_chol`.

Notice that if A_{11} exists within one node, then A_{21} exists within one column of nodes. Recognizing that $A_{21} \leftarrow L_{21} = A_{21}L_{11}^{-T}$ is a row-wise operation and thus parallelizes trivially if L_{11} is duplicated within the column that owns L_{11} (and thus A_{21}), a further optimization is attained by duplicating L_{11} within the appropriate column of nodes, and performing the update of A_{21} locally on those nodes. The resulting PLAPACK code is given in Fig. 2 (c).

A new parallel algorithm

To understand further optimizations, one once again needs to know more about some of the principles underlying PLAPACK. Unlike ScaLAPACK, PLAPACK has a distinct approach to distributing vectors. For scalability reasons, parallel dense linear algebra algorithms must be implemented using a logical two-dimensional mesh of nodes [5, 11, 13, 15]. However, for the distribution of vectors, that two-dimensional mesh is linearized by ordering the nodes in column-major order. Assignment of a vector to the nodes is then accomplished by partitioning the vector in blocks and wrapping these blocks onto the linear array of nodes in round-robin fashion. Assignment of matrices is then dictated by the following principle:

- the ith *row* of the matrix is assigned to the same *row* of nodes as the ith element of the vector.
- the jth *column* of the matrix is assigned to the same *column* of nodes as the jth element of the vector.

In [8, 16] we call this approach to generating a matrix distribution from a vector distribution *physically based matrix distribution*. A consequence of this assignment strategy is that a column of a matrix can be redistributed like a vector by *scattering* the elements appropriately within rows of nodes. Similarly, a row of a matrix can be redistributed like a vector by *scattering* the elements appropriately within columns of nodes. Naturally, redistributing a vector like a column or row of a matrix can be accomplished by reversing these communications.

Note that as a consequence of the last optimization, all computation required to factor A_{11} and update A_{21} is performed within one column of nodes. This is an operation that is very much in the critical path, and thus contributes considerably to the overall time required for the Cholesky factorization. If one views the panel of the matrix

$$\left(\frac{A_{11}}{A_{21}} \right)$$

as a collection (*panel*) of columns, then by simultaneously scattering the elements of these columns of matrices within rows of nodes one can redistribute the panel as a collection of vectors (*multivector*). Subsequently, A_{11} can be factored as a multivector and, more importantly, considerably more parallelism can be attained during the update of A_{21}, since the multivector is distributed to nodes using a one-dimensional data distribution.

It should be noted that the volume of data communicated when redistributing (scattering or gathering) a panel of length n and width b is $O(\frac{n}{r}b)$, where r equals the number of rows in the mesh of nodes. The cost of (parallel) computation subsequently performed on the panel is $O(\frac{n}{r}b^2)$. Thus, if the block size b is relatively large, there is an advantage to redistributing the panel. Furthermore, the multivector distribution is a natural intermediate distribution for the data movement that subsequently must be performed to duplicate the data as part of the parallel symmetric rank-k update [16]. A further optimization of the parallel Cholesky factorization, for which we do *not* present code in Fig. 2, takes advantage of this fact.

Naturally, one may get the impression that we have merely hidden all complexity and ugliness in routines like `PLA_Chol_mv`. Actually, each of those routines themselves are coded at the same high level, and are *not* substantially more complex than the examples presented.

3 LU Factorization (with pivoting)

Next, we discuss the computation of the LU factorization

$$PA = LU$$

where A is an $m \times n$ matrix, L is an $m \times n$ lower trapezoidal matrix with unit diagonal, and U is an $n \times n$ uppertriangular matrix. P is an $m \times m$ permutation matrix which has the property that $P = P_{n-1} \cdots P_1$ where P_j is the permutation matrix that swaps the jth row with the pivot row during the jth iteration of the outerloop of the algorithm.

3.1 Basic algorithm

Let us assume that we have already computed permutations $P_1, \ldots, P_j$ such that

$$P_j \cdots P_1 A = \left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array} \right) = \left(\begin{array}{c} L_{TL} \\ \hline\hline L_{BL} \end{array} \right) \left(\begin{array}{c||c} U_{TL} & U_{TR} \end{array} \right) + \left(\begin{array}{c||c} 0 & 0 \\ \hline\hline 0 & A_{BR} - L_{BR} U_{TL} \end{array} \right)$$

where A_{TL}, L_{TL}, and U_{TL} are $j \times j$, and the matrix has been overwritten by

$$\left(\begin{array}{c||c} L \backslash U_{TL} & U_{TR} \\ \hline\hline L_{BL} & A_{BR} - L_{BL} U_{TR} \end{array} \right)$$

The outer-product based variant, implemented for example in LAPACK, for computing the LU factorization with partial pivoting proceeds as follows:

- Partition A_{BR} (which has been overwritten as described above) like

$$A_{BR} = \left(\begin{array}{c|c} a_{B1} & A_{B2} \end{array} \right)$$

  where a_{B1} equals the first column of A_{BR}.
- Compute permutation $\hat{P}_{j+1}$ which swaps the first and $(j+1)$st rows of A_{BR}, where the $(j+1)$st element in a_{B1} equals the element with maximal absolute value in that vector.
- Permute $\left(\begin{array}{c|c} A_{BL} & A_{BR} \end{array} \right) \leftarrow \hat{P}_{k+1} \left(\begin{array}{c|c} A_{BL} & A_{BR} \end{array} \right)$ and repartition updated A_{BR} like

$$A_{BR} = \left(\begin{array}{c|c} \alpha_{11} & a_{12}^T \\ \hline a_{21} & A_{22} \end{array} \right)$$

- Update $a_{21} \leftarrow l_{21} / \alpha_{11}$.
- Update $A_{22} \leftarrow A_{22} - l_{21} a_{12}^T$.
- Repartition

$$P_{k+1} P_k \cdots P_1 A = \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array} \right)$$

  where A_{TL} is now $(j+1) \times (j+1)$, and

$$P_{j+1} = \left(\begin{array}{c|c} I_j & 0 \\ \hline 0 & \hat{P}_{j+1} \end{array} \right)$$

- Continue with this new partitioning.

It can be easily shown that after the above described steps

$$P_{j+1}P_j\cdots P_1 A = \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c} L_{TL} \\ \hline L_{BL} \end{array}\right) \left(\begin{array}{c|c} U_{TL} & U_{TR} \end{array}\right) + \left(\begin{array}{c|c} 0 & 0 \\ \hline 0 & A_{BR} - L_{BR}U_{TL} \end{array}\right)$$

where A_{TL}, L_{TL}, and U_{TL} are now $(j+1)\times(j+1)$, and the matrix has been overwritten by

$$\left(\begin{array}{c|c} L\backslash U_{TL} & U_{TR} \\ \hline L_{BL} & A_{BR} - L_{BL}U_{TR} \end{array}\right)$$

Notice that when $j=0$, the matrix contains the original matrix, while when $j=n$, it has been overwritten with the L and U factors.

3.2 Blocked algorithm

Given the basic algorithm described above, it is not difficult to derive a blocked algorithm as we show next. The result is given in Fig. 3.

3.3 PLAPACK Implementation

A new parallel algorithm

The implementation of the LU factorization in parallel using PLAPACK closely follows the blocked algorithm discussed in the last section. There are, however, two important differences.

First, pivoting is more complex in parallel. The pivot is determined by a straightforward call to `PLA_Iamax()`. However, the permutation of rows may require communication. Second, on a parallel machine with cartesian deistribution, the matrix-vector-based factoring of the left panel of columns of the matrix may incur load imbalance (because the panel only exists within a single column of processors.) We circumvent this load imbalance by redistributing the panel as a PLAPACK multivector during the panel factorization, as discussed in the section on Cholesky factorization.

With these points in mind, we present the algorithm for the parallel LU factorization side by side with the PLAPACK code implementation. The code for the column-by-column factorization of the panel as multivector is not presented. It is a straightforward translation of the algorithm presented in Section 3.3.

4 Householder QR Factorization

In this section, we discuss the computation of the QR factorization

$$A = QR$$

where A is $m\times n$, Q is $m\times m$ and R is $m\times n$. Here $m\geq n$, Q is unitary ($QQ^T = Q^TQ = I$) and R has the form

$$R = \left(\begin{array}{c} R_1 \\ \hline 0 \end{array}\right)$$

where R_1 is an $n\times n$ uppertriangular matrix. Partitioning

$$Q = \left(\begin{array}{c|c} Q_1 & Q_2 \end{array}\right)$$

where Q_1 has width n, we see that the following also holds

$$A = Q_1R_1$$

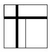
In our subsequent discussions, we will refer to both of these factorizations as a QR factorization and will explicitly indicate which of the two is being considered.

partition $A = \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$
where A_{TL} is 0×0

do until A_{BR} is 0×0

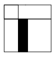
determine block size b

partition



$$\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c||cc} A_{TL} & \multicolumn{2}{c}{A_{TR}} \\ \hline\hline A_{BL} & A_{B1} & \multicolumn{1}{|c}{A_{B2}} \end{array}\right)$$

where A_{B1} has width b

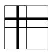


factor $P_{\mathrm{cur}} A_{B1} \rightarrow \left(\begin{array}{c} L_{11} \\ \hline L_{21} \end{array}\right) \quad U_{11}$

$\left(A_{B0} \,\|\, A_{B1} \,|\, A_{B2} \right)$
$\leftarrow P_{\mathrm{cur}} \left(A_{B0} \,\|\, A_{B1} \,|\, A_{B2} \right)$

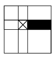
overwrite A_{B1} with factored panel

repartition

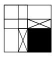


$$\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c||c|c} A_{00} & A_{01} & A_{02} \\ \hline\hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

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

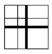


$A_{12} \leftarrow L_{12} = L_{11}^{-1} A_{12}$



$A_{22} \leftarrow A_{22} - L_{21} L_{12}$

continue with



$$\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c|c||c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline\hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

enddo

(a) Blocked algorithm

```
PLA_Obj_view_all( A, &A_B );
PLA_Obj_view_all( A, &ABR );

while ( TRUE ) {
  PLA_Obj_global_length( ABR , &b );
  b = min( b, nb_alg );
  if ( 0 == b ) break;

  PLA_Obj_vert_split_2( ABR, b, &AB1, &AB2 );
  PLA_Obj_vert_split_2 ( pivots, b,
                         &pivot_1, &pivots);

  PLA_Mvector_create_conf_to( AB1, b, &AB1_mv );
  PLA_Copy ( AB1, AB1_mv);
  LU_mv ( AB1_mv, pivot_1);


  Apply_pivots ( A_B, pivot_1);

  PLA_Copy ( A_1_mv, A_1);


  PLA_Obj_split_4( ABR, b, b, &A11, &A12,
                              &A21, &ABR );


  PLA_Trsm( PLA_SIDE_LEFT, PLA_LOW_TRIAN,
            PLA_NO_TRANS, PLA_UNIT_DIAG,
            one, A11, A21 );
  PLA_Gemm( PLA_NO_TRANS, PLA_NO_TRANS,
            minus_one, A21, A12, one, ABR );

  PLA_Obj_horz_split_2( A_B, b,  PLA_DUMMY,
                                 &A_B );
}
```

(b) PLAPACK implementation

Figure 3: The blocked LU factorization algorithm in PLAPACK.

4.1 Basic algorithm

To present a basic algorithm for the QR factorization, we must start by introducing Householder transforms (reflections).

Householder transforms are orthonormal transformations that can be written as

$$H = (I + \beta v v^T)$$

where $\beta = -2||v||_2^2$. These transformations, sometimes called reflectors, have a number of interesting properties: $H^T = H$, $H^T H = H H^T = HH = I$, and $||H||_2 = 1$. Of most interest to us is the fact that given a vector $x = \left(x_1^T, x_2^T\right)^T$, where x_1 is of length $k-1$, one can find a vector v_k such that $P_k x = \left(x_1^T, \pm||x_2||_2 e_1^T\right)$ where $e_1 = (1, 0, \cdots, 0)^T$. Indeed, it can be easily verified that

$$v_k = \left(\begin{array}{c} 0 \\ \vdots \\ 0 \\ \hline x_2 \mp ||x_2||_2 e_1 \end{array} \right)$$

has this property. Notice that k here is used to indicate that the first $k-1$ elements of x are to be left alone, which results in v_k having $k-1$ zero valued leading elements. The sign that is chosen corresponds to the sign of the first entry of x_2 to avoid catastrophic cancellation. Vector v_k can be scaled arbitrarily by adjusting β correspondingly. In the subsequent section, we will scale it so that the first nonzero (kth) entry of v_k is 1.

To compute the QR factorization of given $m \times n$ matrix A, we wish to compute Householder transformations $H_1, \ldots, H_n$ such that

$$H_n \cdots H_1 A = R$$

where R is uppertriangular.

An algorithm for computing the QR factorization is given by

1. Partition $A = \left(\begin{array}{c|c} a_1 & A_2 \end{array} \right)$
2. Determine (v, β) corresponding to the vector a_1.
3. Store v in the now zero part of a_1.
4. $A_2 \leftarrow (I + \beta v v^T) A_2 = A_2 + \beta v y^T$ where $y^T = v^T A_2$
5. Repartition $A = \left(\begin{array}{c|c} \alpha_{11} & a_{12}^T \\ \hline a_{21} & A_{22} \end{array} \right)$
6. Continue recursively with A_{22}

4.1.1 Blocked algorithm

In [3, 9], it is shown how a blocked (matrix-matrix multiply) based algorithm can be derived , by creating a WY transform, which, when applied, yield the same result as a number of successive applications of Householder transforms:

$$I + WY^T = H_1 H_2 \cdots H_k$$

where W and Y are both $n \times k$. Given the k vectors that define the k Householder transforms, these matrices can be computed one column at a time in a relatively straight forward manner.

Given the WY transform discussion above, the blocked version of the algorithm is given in Fig. 4.

4.2 PLAPACK Implementation

A new parallel algorithm

The expert optimization of the QR factorization lies in the ability of PLAPACK of viewing parts of the matrix as a panel that can be redistributed as a multivector on the nodes. A PLAPACK implementation that explores this is given in Fig. 4.

partition $A = \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array} \right)$

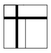
where A_{TL} is 0×0

partition $\beta = \left(\begin{array}{c} \beta_T \\ \hline \beta_B \end{array} \right)$

where β_T is of length 0

do until A_{BR} has width 0

determine block size b



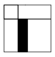
partition

$$\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array} \right) = \left(\begin{array}{c||c|c} A_{TL} & \multicolumn{2}{c}{A_{TR}} \\ \hline\hline A_{BL} & A_{B1} & A_{B2} \end{array} \right)$$

where A_{B1} has width b

partition

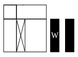
$$\left(\begin{array}{c} \beta_T \\ \hline\hline \beta_B \end{array} \right) = \left(\begin{array}{c} \beta_0 \\ \hline\hline \beta_1 \\ \hline \beta_2 \end{array} \right) \text{ where } \beta_1 \text{ has length } b$$



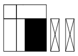
$A_{B1} \rightarrow H_1 \cdots H_B R_{11}$

computing only the vectors for $H_1, \ldots, H_b$ and storing scaling factors in β_1

copy R_{11} and Householder vectors to A_{B1}

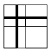


compute W and Y from A_{B1} and β_1



$A_{B2} \leftarrow (I + WY^T) A_{B2}$

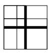
repartition



$$\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array} \right) = \left(\begin{array}{c||c|c} A_{00} & A_{01} & A_{12} \\ \hline\hline A_{10} & A_{11} & A_{12} \\ \hline A_{30} & A_{31} & A_{32} \end{array} \right)$$

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

continue with



$$\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array} \right) = \left(\begin{array}{c|c||c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline\hline A_{20} & A_{21} & A_{22} \end{array} \right)$$

and $\left(\begin{array}{c} \beta_T \\ \hline\hline \beta_B \end{array} \right) = \left(\begin{array}{c} \beta_0 \\ \hline \beta_1 \\ \hline\hline \beta_2 \end{array} \right)$

enddo

(a) Blocked algorithm

```
PLA_Obj_view_all( A, &ABR );


PLA_Obj_view_all( beta, &beta_B );


while ( TRUE ) {
  PLA_Obj_global_length( ABR , &b );
  if ( ( b = min( b, nb_alg ) ) == 0 ) break;


  PLA_Obj_vert_split_2( ABR, b,    &AB1, &AB2 );


  PLA_Obj_horz_split_2(
                  beta_B, b, &beta1, &beta_B );


  PLA_Mvector_create_conf_to( AB1, b, &AB1_mv );
  PLA_Copy( AB1, AB1_mv );
  PLA_QR_mv( AB1_mv, beta1 );

  PLA_Copy( AB1_mv, AB1 );

  PLA_Mvector_create_conf_to( AB1, b, &W_mv );
  PLA_Mvector_create_conf_to( AB1, b, &Y_mv );
  PLA_Compute_WY( AB1_mv, beta1, W_mv, Y_mv );

  PLA_Apply_W_Y_transform ( PLA_SIDE_LEFT,
               PLA_NO_TRANS, W_mv, Y_mv, AB2);


  PLA_Obj_horz_split_2( AB2, b,      PLA_DUMMY,
                                     &ABR );
}
```

(b) PLAPACK implementation

Figure 4: Optimized implementation of QR factorization using PLAPACK

5 Performance

In this section, we present performance data measures on a Cray T3E-600. While we have run the algorithms on many different machines, and many different numbers of nodes, the presented data was chosen since it is the most complete data for all different algorithms.

In our performance graphs in Fig. 5, we report millions of floating point operations per second (MFLOPS) *per node* attained by the algorithms. For the Cholesky, LU, and QR factorizations, we use the accepted operation counts of $\frac{1}{3}n^3$, $\frac{2}{3}n^3$, and $\frac{4}{3}n^3$, respectively. (Notice that we only measured performance for square matrices.) All computation was performed in 64-bit arithmetic. The LINPACK benchmark (essentially an LU factorization) on the same architecture and configuration attains close to 400 MFLOPS per node, but for a much larger problem (almost 20000x20000) than we measured.

We compare performance of our implementations to performance attained by the routines with the same functionality provided by ScaLAPACK. The presented curves for ScaLAPACK were created with data from the ScaLAPACK Users' Guide [4]. While it is highly likely that that data was obtained with different versions of the OS, MPI, compilers, and BLAS, we did perform some measurements of ScaLAPACK routines on the same system on which our implementations were measured. The data in the ScaLAPACK Users' Guide was found to be representative of performance on our system.

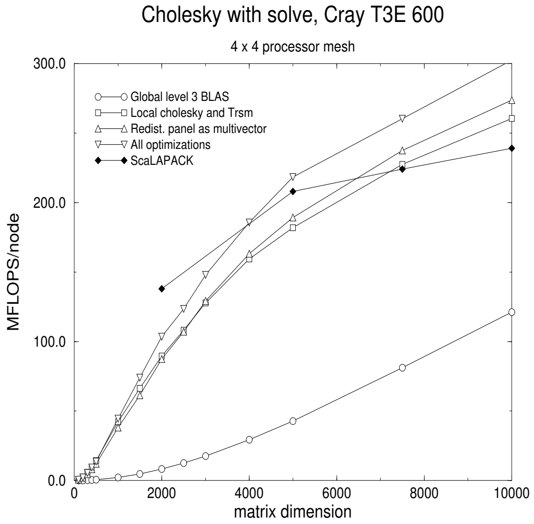
In Fig. 5 (a), we present the performance of the various parallel Cholesky implementations presented in Section 2. The following table links the legend for the curves to the different implementations:

Curve	Algorithm	`nb_alg`	`nb_distr`
`Global level 3 BLAS`	Fig. 2 (b)	128	64
`Local Cholesky and Trsm`	Fig. 2 (c)	64	64
`Redist. panel as multivector`	Fig. 2 (d)	128	64
`All optimizations`	Further optimization at end of Section 2.3	128	64
`ScaLAPACK`	As reported in the ScaLAPACK Users' Guide	32	32

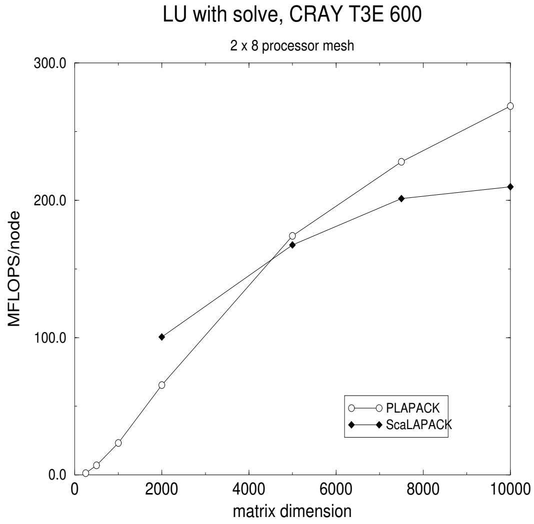
Recall that the basic implementation `Global level 3 BLAS` calls an unblocked parallel implementation of the Cholesky factorization, which incurs considerablein turn communication overhead. It is for this reason that performance is quite disappointing. Picking the block size so that this subproblem can be solved on a single node, and the corresponding triangular solve with multiple right-hand-sides on a single column of nodes, boosts performance to quite acceptable levels. Our new algorithm, which redistributes the panel to be factored before factoring, improves upon this performance somewhat. Final minor further optimizations, not described in this paper, boost the performance past 300 MFLOPS per node for problem sizes that fit our machine.

In Fig. 5 (b)–(c), we present only the "new parallel algorithm" versions of LU factorization and QR factorization. Performance behavior is quite similar to that observed by the Cholesky factorization. It is interesting to note that the performance of the QR factorization is considerably less than that of the Cholesky or LU factorization. This can be attributed to the fact that the QR factorization performs the operation $U = Y^T A_{B2}$ in addition to the rank-k update $A_{B2} \leftarrow A_{B2} + WU^T$. This, in turn, local on each node, calls a different case of the matrix-matrix multiplication kernel, which does not attain the same level of performance in the current implementation of the BLAS for the Cray T3E. We should note that the PLAPACK implementations use an algorithmic block size (`nb_alg`) of 128 and a distribution block size (`nb_distr`) of 64, while ScaLAPACK again uses blocking sizes of 32. Finally, in Fig. 5 (d), we present performance of the LU factorization on different numbers of nodes, which gives some indication of the scalability of the approach.

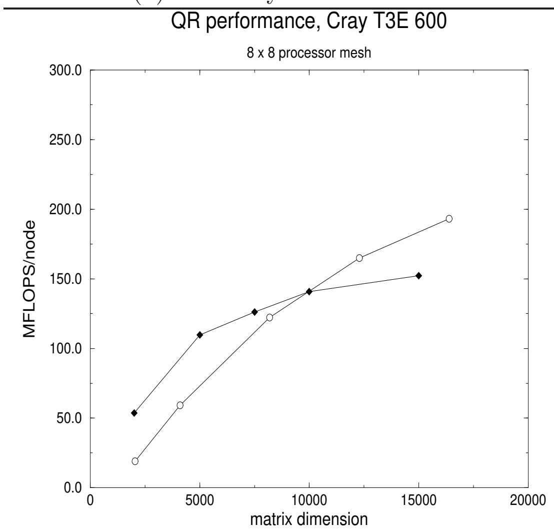
When comparing performance of PLAPACK and ScaLAPACK, it is clear that the PLAPACK overhead at this point does impose a penalty for small problem sizes. The algorithm used for Cholesky factorization by ScaLAPACK is roughly equivalent to the one marked `Local Cholesky and Trsm` in the graph. Even for the algorithm that is roughly equivalent to the ScaLAPACK implementation, PLAPACK outperforms ScaLAPACK for larger problem sizes. This is in part due to the fact that we use a larger algorithmic and distribution block size (64 vs. 32) for that algorithm. It is also due to the fact that we implemented the symmetric rank-k update (`PLA_Syrk`) differently from the ScaLAPACK



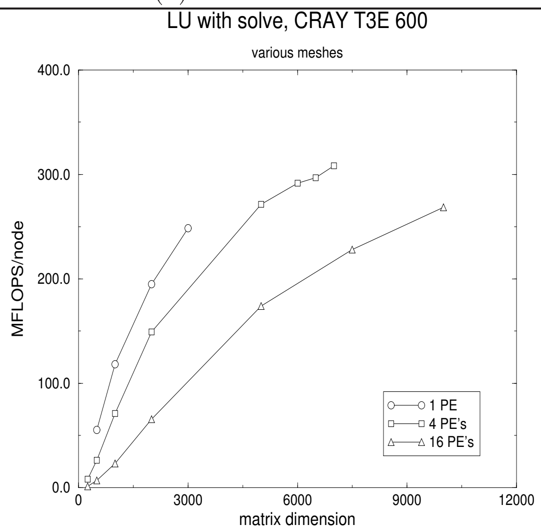
(a) Cholesky factorization



(b) LU factorization



(c) QR factorization



(d) LU factorization on different configurations

Figure 5: Performance of the various algorithms on a 16 processor configuration of the Cray T3E 600.

equivalent routine (`PSSYRK`). We experimented with using an algorithmic and distribution block size of 64 for ScaLAPACK, but this did not improve performance. Unlike the more sophisticated algorithms implemented using PLAPACK, ScaLAPACK does not allow the algorithmic and distribution block size to differ.

6 Conclusion

In this paper, we have presented algorithms for the Cholesky, LU, and QR factorizations and shown how they can be implemented for distributed memory parallel architectures at a high level of abstraction. Although at this moment there is a considerable overhead for this high level abstraction, for large enough problems this is not as noticeable. Indeed, by implementing more ambitious algorithms, considerable performance gains can be made, when compared to more traditional approaches. We believe that by optimizing the underlying infrastructure, the overhead can be greatly reduced, and thus even for smaller problems high performance will be attained.

Acknowledgments

We are indebted to the rest of the PLAPACK team (in particular, Philip Alpatov, Dr. Carter Edwards, James Overfelt, and Dr. Yuan-Jye Jason Wu) for providing the infrastructure that made this research possible.

Primary support for this project come from the Parallel Research on Invariant Subspace Methods (PRISM) project (ARPA grant P-95006) and the Environmental Molecular Sciences construction project at Pacific Northwest National Laboratory (PNNL) (PNNL is a multiprogram national laboratory operated by Battelle Memorial Institute for the U.S. Department of Energy under Contract DE-AC06-76RLO 1830). Additional support for PLAPACK came from the NASA High Performance Computing and Communications Program's Earth and Space Sciences Project (NRA Grants NAG5-2497 and NAG5-2511), and the Intel Research Council.

We gratefully acknowledge access to equipment provided by the National Partnership for Advanced Computational Infrastructure (NPACI), The University of Texas Computation Center's High Performance Computing Facility, and the Texas Institute for Computational and Applied Mathematics' Distributed Computing Laboratory.

Additional information

For additional information, visit the PLAPACK web site:

`http://www.cs.utexas.edu/users/plapack`

References

[1] Philip Alpatov, Greg Baker, Carter Edwards, John Gunnels, Greg Morrow, James Overfelt, Robert van de Geijn, and Yuan-Jye J. Wu. Plapack: Parallel linear algebra package – design overview. In *Proceedings of SC97*, 1997.

[2] E. Anderson, Z. Bai, J. Demmel, J. E. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. E. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, 1992.

[3] Christian Bischof and Charles Van Loan. The WY representation for products of Householder matrices. *SIAM J. Sci. Stat. Comput.*, 8(1):s2–s13, Jan. 1987.

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

[5] Jack Dongarra, Robert van de Geijn, and David Walker. Scalability issues affecting the design of a dense linear algebra library. *J. Parallel Distrib. Comput.*, 22(3), Sept. 1994.

[6] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 16(1):1–17, March 1990.

[7] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 14(1):1–17, March 1988.

[8] C. Edwards, P. Geng, A. Patra, and R. van de Geijn. Parallel matrix distributions: have we been doing it all wrong? Technical Report TR-95-40, Department of Computer Sciences, The University of Texas at Austin, 1995.

[9] Gene H. Golub and Charles F. Van Loan. *Matrix Computations.* The Johns Hopkins University Press, Baltimore, second edition, 1989.

[10] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI.* The MIT Press, 1994.

[11] B. A. Hendrickson and D. E. Womble. The torus-wrap mapping for dense matrix calculations on massively parallel computers. *SIAM J. Sci. Stat. Comput.*, 15(5):1201–1226, 1994.

[12] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Soft.*, 5(3):308–323, Sept. 1979.

[13] W. Lichtenstein and S. L. Johnsson. Block-cyclic dense linear algebra. Technical Report TR-04-92, Harvard University, Center for Research in Computing Technology, Jan. 1992.

[14] Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI: The Complete Reference.* The MIT Press, 1996.

[15] G.W. Stewart. Communication and matrix computations on large message passing systems. *Parallel Computing*, 16:27–40, 1990.

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