# PLAPACK: Parallel Linear Algebra Package*

Philip Alpatov † Greg Baker † Carter Edwards † John Gunnels †
Greg Morrow † James Overfelt † Robert van de Geijn †‡
Yuan-Jye J. Wu §

**Abstract**

The PLAPACK project represents an effort to provide an infrastructure for implementing application friendly high performance linear algebra algorithms. The package uses a more application-centric data distribution, which we call Physically Based Matrix Distribution, as well as an object based (MPI-like) style of programming. It is this style of programming that allows for highly compact codes, written in C but usable from FORTRAN, that more closely reflect the underlying blocked algorithms. We show that this can be attained without sacrificing high performance.

## 1 Introduction

Parallel implementation of most dense linear algebra operations is a relatively well understood process. Nonetheless, availability of general purpose, high performance parallel dense linear algebra libraries is severely hampered by the fact that translating the sequential algorithms, which typically can be described without filling up more than half a chalk board, to a parallel code requires careful manipulation of indices and parameters describing the data, its distribution to processors, and/or the communication required. It is this manipulation of indices that easily leads to bugs in parallel code. This in turn stands in the way of the parallel implementation of more sophisticated algorithms, since the coding effort simply becomes overwhelming.

The Parallel Linear Algebra Package (PLAPACK) infrastructure attempts to overcome this complexity by providing a coding interface that mirrors the natural description of sequential dense linear algebra algorithms. To achieve this, we have adopted an "object based" approach to programming. This object based approach has already been popularized for high performance parallel computing by libraries like the Toolbox being developed at Mississippi State University [3], the PETSc library at Argonne National Laboratory [2], and the Message-Passing Interface [8].

The PLAPACK infrastructure uses a data distribution that starts by partitioning the vectors associated with the linear algebra problem and assigning the sub-vectors to

*This project was sponsored in part by the Parallel Research on Invariant Subspace Methods (PRISM) project (ARPA grant P-95006), the NASA High Performance Computing and Communications Program's Earth and Space Sciences Project (NRA Grants NAG5-2497 and NAG5-2511), 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), and the Intel Research Council.

†The University of Texas, Austin, TX 78712
‡rvdg@cs.utexas.edu
§Argonne National Laboratory, Argonne, IL 60439, jwu@mcs.anl.gov

nodes. The matrix distribution is then *induced* by the distribution of these vectors. This approach was chosen in an attempt to create more reasonable interfaces between applications and libraries. However, the surprising discovery has been that this approach greatly *simplifies* the implementation of the infrastructure, allowing much more generality (in future extensions of the infrastructure) while reducing the amount of code required when compared to previous generation parallel dense linear algebra libraries [4].

In this paper, we primarily concentrate on giving the reader a flavor of what it is like to code using the PLAPACK infrastructure.

## 2 Natural Description of Linear Algebra Algorithms

Let us consider the simple example of implementation of the Cholesky factorization. Given a symmetric $n \times n$ positive definite matrix $A$, the Cholesky factorization of $A$ is given by

$$A = LL^T$$

where $L$ is lower triangular.

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

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

where $a_{11}$ and $l_{11}$ are scalars, and $a_{21}$ and $l_{21}$ are vectors of length $n-1$. The $\star$ indicates the symmetric part of $A$. Now,

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

This in turn yields the equations

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

We now give the algorithm as it might appear on a chalk board in a classroom, and the PLAPACK implementation:

<table>
<tr>
<td>

let $A_{\text{cur}} = A$

do until $A_{\text{cur}}$ is $0 \times 0$

Partition $A_{\text{cur}} = \left( \begin{array}{c|c} a_{11} & \star \\ \hline a_{21} & A_{22} \end{array} \right)$

$a_{11} \leftarrow l_{11} = \sqrt{a_{11}}$

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

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

continue with $A_{\text{cur}} = A_{22}$

</td>
<td>

```
PLA_Obj_view_all( a, &acur );
while ( TRUE ){
     PLA_Obj_global_length( acur, &size );
     if ( 0 == size ) break;
     PLA_Obj_split_4( acur, 1, 1, &a11, &a12,
                                  &a21, &acur );
     Take_sqrt( a11 );
     PLA_Inv_scal( a11, a21 );
     PLA_Syr( PLA_LOW_TRIAN, min_one, a21, acur );
}
```

</td>
</tr>
</table>

All information that describes $A$, its data, and how it is distributed to processors (nodes) is encoded in a object (data-structure) referenced by `a`. The `PLA_Obj_view_all` creates a second reference, `acur`, into the same data. The `PLA_Obj_global_length` call extracts the current size of the matrix that `acur` currently references. If this is zero, the recursion has completed. The call to `PLA_Obj_split_4` partitions the matrix, creating new references for the four quadrants. Element $a_{11}$ is updated by taking its square root in subroutine `Take_sqrt`, $a_{21}$ is scaled by $1/a_{11}$ by calling `PLA_Inv_scal`, and finally the symmetric rank-1 update is achieved through the call to `PLA_Syr`. This sets up the next iteration, which is also the next level of recursion in our original algorithm.

## 3 Physically Based Matrix Distribution

We postulate in [7] that one should never start by considering how to decompose the matrix. Rather, one should start by considering how to decompose the physical problem to be solved. Notice that it is the *elements of vectors* that are typically associated with data of physical significance and it is therefore their distribution to nodes that is directly related to the distribution of the problem to be solved. A matrix (discretized operator) merely represents the relation between two vectors (discretized spaces): $y = Ax$. Since it is more natural to start with distributing the problem to nodes, we partition $x$ and $y$, and assign portions of these vectors to nodes. The matrix $A$ should then be distributed to nodes in a fashion consistent with the distribution of the vectors, as we shall show next. We will call a matrix distribution *physically based* if the layout of the vectors which induce the distribution of $A$ to nodes is consistent with where an application would naturally want them. We will use the abbreviation *PBMD* for *Physically Based Matrix Distribution.*

As discussed, we must start by describing the distribution of the vectors, $x$ and $y$, to nodes, after which we will show how the matrix distribution is *induced* (derived) by the vector distribution. In the current implementation of PLAPACK, the distribution is derived as follows: partitioning $x$ and $y$ so that

$$x = \left( \begin{array}{c} x_0 \\ \hline x_1 \\ \hline \vdots \\ \hline x_{N-1} \end{array} \right) \text{ and } y = \left( \begin{array}{c} y_0 \\ \hline y_1 \\ \hline \vdots \\ \hline y_{N-1} \end{array} \right)$$

where $N >> p$. Partitioning $A$ conformally yields the blocked matrix

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

An explicitly two-dimensional wrapped distribution for the matrix can now be obtained by wrapping the blocks of $x$ and $y$ onto a two-dimensional mesh of nodes in column-major order, and assigning blocks of columns $A_{*,j}$ to the same column of nodes as sub-vector $x_j$, and block or rows $A_{i,*}$ to the same row of nodes as sub-vector $y_i$. For details, see [7, 9].

We need to clearly state that in the *initial* implementation of PLAPACK, we only implement this special case. Generalizations are planned.

## 4 Linear Algebra Objects

In Section 2, we introduced the concept of encoding all information about a matrix in an object. Similarly, PLAPACK supports a number of *linear algebra objects*, including *matrices*, *vectors*, *multivectors* (a collection of vectors), and *projected* vectors and multivectors.

We have already discussed how vectors are distributed. Multivectors are a collection of vectors that are identically distributed. Alternatively, one can think of a multivector as a matrix with a few columns, where each column is distributed like a vector.

Many parallel linear algebra libraries view vectors as special cases of matrices, which has the consequence that they exist within on column of nodes or one row of nodes. We also include this case, but use it only for intermediate results. Thus, we view these cases of vectors as vectors that have been *projected* onto a row of nodes, gathering the elements of the vector within each column of nodes to the node in the desired row. Similarly, a vector can be projected onto a column by gathering the elements of the vector within each row of nodes to the node in the desired column. Similarly, multivectors can be projected.

Finally, a vector that is projected onto a row of nodes can be duplicated to all rows of nodes, created a duplicated (row) vector. This can be achieved by collecting (instead of gathering) within each column of nodes. Similarly, a duplicated projected column vector can be created. Extensions to multivectors naturally follow.

There is one additional linear algebra object, a multiscalar. A multiscalar is a scalar or matrix that is not distributed (i.e., it exists entirely within on node), although it can be duplicated to all nodes.

## 5 Implementation of Matrix-Matrix Multiplication

Perhaps the best way to illustrate how the PLAPACK infrastructure can be used to code various parallel linear algebra algorithms is in detail discuss a reasonable simple example. For this we choose the parallel implementation of $C = AB$, where in our discussion we assume all three matrices are $n \times n$.

### 5.1 Algorithm

Computation of $C = AB$ can be implemented by partitioning

$$A = \left( \begin{array}{c|c|c} A_0 & A_1 & \cdots \end{array} \right) \quad \text{and} \quad B = \left( \begin{array}{c} B_0 \\ \hline B_1 \\ \hline \vdots \end{array} \right)$$

and noticing that

$$C = AB = A_0 B_0 + A_1 B_1 + \cdots$$

Thus the implementation of this operation can proceed as a sequence of *rank-k updates* [1, 5, 10].

Let us concentrate of one update: $C = C + A_k B_k$. Partitioning these matrices as they were when we discussed distribution of matrices yields

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

$$A_k = \left( \begin{array}{c} A_{0,k} \\ \hline A_{1,k} \\ \hline \vdots \\ \hline A_{N-1,k} \end{array} \right) \quad B_k = \left( \begin{array}{c|c|c|c} B_{k,0} & B_{k,1} & \cdots & B_{k,N-1} \end{array} \right)$$

so that $C_{i,j} = C_{i,j} + A_{i,k}B_{k,j}$. Careful consideration shows that if the matrices are appropriately aligned, then duplicating $A_{*,k}$ within rows of nodes and $B_{k,*}$ within columns of nodes will allow local rank-k updates to proceed.

The more general case $C = \alpha AB + \beta C$ can be implemented using the following recursive algorithm:

scale $C \leftarrow \beta C$
let $A_{\text{cur}} = A$ and $B_{\text{cur}} = B$
do until done:

$$A_{\text{cur}} = \left( \begin{array}{c|c} A_1 & A_2 \end{array} \right) \quad \text{and} \quad B_{\text{cur}} = \left( \begin{array}{c} B_1 \\ \hline B_2 \end{array} \right)$$

$C \leftarrow \alpha A_1 B_1 + C$
let $A_{\text{cur}} = A_2$ and $B_{\text{cur}} = B_2$

## 5.2 PLAPACK implementation

The PLAPACK implementation is given in Fig. 1. We explain the code line-by-line:

| lines | |
|---|---|
| **3-4** | The matrices and constants are passed in linear algebra objects `alpha`, `a`, `b`, `beta`, and `c`. |
| **8:** | Scale $C \leftarrow \beta C$ |
| **14-16:** | Extract the datatype of the objects and create a $1 \times 1$ multiscalar `one` that is duplicated to all nodes. |
| **18-19:** | $A_{\text{cur}} = A$ and $B_{\text{cur}} = B$. |
| **20-22:** | Create *duplicated* projected multivectors to hold $A_1$ and $B_1$ after duplication within rows and columns, respectively. |
| **24-28:** | Loop until the current matrices are empty. |
| **28:** | Determine the width of $A_1$ and height of $B_1$. |
| **30-33:** | $A_{\text{cur}} = \left( \begin{array}{c\|c} A_1 & A_2 \end{array} \right)$ and $B_{\text{cur}} = \left( \begin{array}{c} B_1 \\ \hline B_2 \end{array} \right)$ |
| **35-37:** | Size the duplicated projected multivectors appropriately. |
| **39-40:** | Indicate that $A_1$ and $B_1$ are oriented similar to multivectors projected onto a column and row of nodes, respectively. |
| **42-43:** | Duplicate $A_1$ and $A_2$. Notice that the input and output objects are described, after which *all* communication is hidden in this copy. |
| **45:** | Perform updates to the local part of $C$, using the duplicated information. |
| **46:** | Continue with $A_{\text{cur}} = A_2$ and $B_{\text{cur}} = B_2$. |
| **47-49:** | Cleanup of temporary objects. |

Naturally, we are only trying to give a flavor of how such algorithms are implemented. For an example of a much more complex algorithm, the parallel implementation of reduction to banded form, see [11].

```
int pgemm_ab_panpan( int nb_alg, PLA_Obj alpha, PLA_Obj a, PLA_Obj b,
                                PLA_Obj beta, PLA_Obj c )
{
  PLA_Obj acur = NULL,     a1 = NULL,    a1dup = NULL,    a1dup_cur = NULL,
          bcur = NULL,     b1 = NULL,    b1dup = NULL,    b1dup_cur = NULL,
          one = NULL;
  MPI_Datatype  datatype;
   int size, width_a1, length_b1;
                                                            /* scale C by beta */
  PLA_Scal( beta, c );
                                                  /* create constant mscalar 1 */
  PLA_Obj_datatype( a, &datatype)
  PLA_Mscalar_create( datatype, PLA_ALL_ROWS, PLA_ALL_COLS, 1, 1, &one );
  PLA_Obj_set_to_one( one );
                                         /* take a view of all of both A and B */
  PLA_Obj_view_all( a,          &acur );
  PLA_Obj_view_all( b,          &bcur );
                           /* create dupl. pmvectors for spreading panels of A and B */
  PLA_Pmvector_create_conf_to( c, PLA_PROJ_ONTO_COL, PLA_ALL_COLS, nb_alg, &a1dup );
  PLA_Pmvector_create_conf_to( c, PLA_PROJ_ONTO_ROW, PLA_ALL_ROWS, nb_alg, &b1dup );
                                               /* Loop until no more of A and B */
  while ( TRUE ) {
                                              /* determine width of next update */
    PLA_Obj_width ( acur, &width_a1 );
    PLA_Obj_length( bcur, &length_b1 );
    if ( ( size = min3( width_a1, length_b1, nb_alg )) == 0 ) break;
                                              /* split off first column of Acur */
    PLA_Obj_vert_split_2 ( acur, size,            &a1, &acur );
                                                /* split off first row of Bcur */
    PLA_Obj_horz_split_2( bcur, size,            &b1,
                                                 &bcur );
                                                        /* size the workspaces */
    PLA_Obj_vert_split_2 ( a1dup, size,          &a1dup_cur, &PLA_DUMMY );
    PLA_Obj_horz_split_2( b1dup, size,          &b1dup_cur,
                                                &PLA_DUMMY );
                                                        /* annotate the views */
    PLA_Obj_set_orientation( a1, PLA_PROJ_ONTO_COL );
    PLA_Obj_set_orientation( b1, PLA_PROJ_ONTO_ROW );
                                                          /* spread a1 and b1 */
    PLA_Copy( a1, a1dup_cur );
    PLA_Copy( b1, b1dup_cur );
                                              /* perform local rank-size update */
    PLA_Local_Gemm( PLA_NOTRANS, PLA_NOTRANS, alpha, a1dup_cur, b1dup_cur, one, c);
  }                                                          /* Free the views */
  PLA_Obj_free( &a1 );    PLA_Obj_free( &a1dup );   PLA_Obj_free( &a1dup_cur );
  PLA_Obj_free( &b1 );    PLA_Obj_free( &b1dup );   PLA_Obj_free( &b1dup_cur );
  PLA_Obj_free( &one );
}
```

FIG. 1. *$C = \alpha AB + \beta C$ using a sequence of rank-nb updates, with nb=*`nb_alg`*, the algorithmic blocking size.*

### 5.3 Performance

We discuss performance on three current generation distributed memory parallel computers: the Intel Paragon with GP nodes (one compute processor per node), the Cray T3D, and the IBM SP-2. In all cases we will report performance per node, where we use the term node for a compute processor. All performance is for 64-bit precision. The peak performance of an individual node limits the peak performance per node that can be achieved for our parallel implementations. The single processor peak performances for 64-bit arithmetic matrix-matrix multiply, using assembly coded sequential BLAS, are roughly given by 46 MFLOPS/sec for the Paragon, 120 MFLOPS/sec for the Cray T3D, and 210 MFLOPS/sec for the IBM SP-2. The speed and topology of the interconnection network affects how fast peak performance can be approached. Since our infrastructure is far from optimized with regards to communication as of this writing, there is no real need to discuss network speeds other than that both the Intel Paragon and Cray T3D have very fast interconnection networks, while the IBM SP-2 has a noticeably slower network, relative to individual processor speeds.

In Figure 2, we show the performance on 64 node configurations of the different architectures. In that figure, the performance of the panel-panel variant is reported, with the matrix dimensions chosen so that all three matrices are square. The algorithmic and distribution blocking sizes, $nb_{\text{alg}}$ and $nb$ were taken to be equal to each other ($nb_{\text{alg}} = nb$), but on each architecture they equal the value that appears to give good performance: on the Paragon $nb_{\text{alg}} = nb = 20$ and on the T3D and SP-2 $nb_{\text{alg}} = nb = 64$. The different curves approach the peak performance for the given architecture, eventually leveling off.

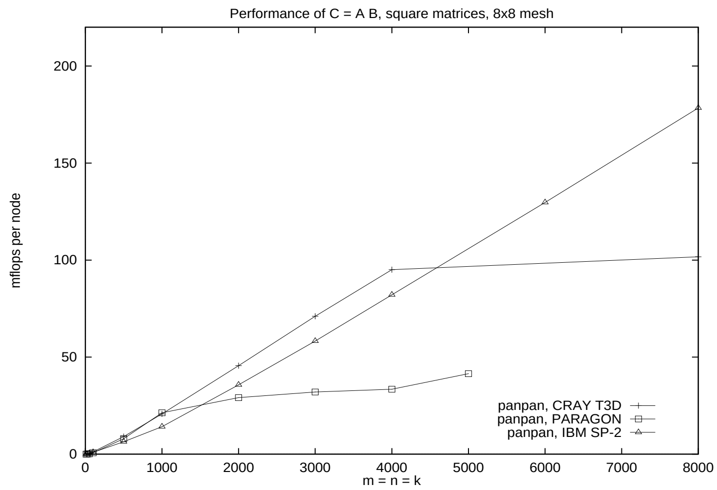


FIG. 2. *Performance of $C = \alpha AB + \beta C$ on 64 processor configurations of the Intel Paragon with GP nodes, the Cray T3D, and the IBM SP-2. The blocking sizes on the Paragon are $nb_{\text{alg}} = nb = 20$ and on the T3D and SP-2 are $nb_{\text{alg}} = nb = 64$.*

## 6 Conclusion

The PLAPACK project attempts to provide an infrastructure that allows a user to quickly implement new algorithms in a fashion that reflects how the algorithm is naturally explained. We believe this supports rapid and robust development of new parallel implementations of dense linear algebra algorithms. Initial performance results indicate that this can be achieved without sacrificing performance. For more information regarding the PLAPACK project, check our web page `http://www.cs.utexas.edu/users/plapack/`. In particular, further performance results are available at that site.

## Acknowledgements

Portions of this paper were taken, with permission, from *Using PLAPACK: Parallel Linear Algebra Package* by R. van de Geijn, published by The MIT Press [9].

Access to various parallel architectures was provided by the following facilities: the University of Texas Computation Center's High Performance Computing Facility, the Texas Institute for Computational and Applied Mathematics' Distributed Computing Laboratory, the Molecular Science Computing Facility in the Environmental Molecular Sciences Laboratory at the Pacific Northwest National Laboratory (PNNL), the Intel Paragon system operated by the California Institute of Technology on behalf of the Concurrent Supercomputing Consortium (Access to this facility was arranged by the California Institute of Technology), and Cray Research, a Silicon Graphics Company.

We would like to thank George Fann at PNNL for collecting the performance data for the IBM SP-2.

## 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. of Research and Development, 38(6) (1994).

[2] S. Balay, W. Gropp, L. Curfman McInnes, and B. Smith, *PETSc 2.0 users manual*, Tech. Report ANL-95/11, Argonne National Laboratory, Oct. 1996.

[3] P. Bangalore, A. Skjellum, C. Baldwin, and S. G. Smith, *Dense and iterative concurrent linear algebra in the multicomputer toolbox*, in Proceedings of the Scalable Parallel Libraries Conference (SPLC '93) (1993), pp. 132–141.

[4] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker, *Scalapack: A scalable linear algebra library for distributed memory concurrent computers*, in Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation, IEEE Comput. Soc. Press (1992), pp. 120–127.

[5] Almadena Chtchelkanova, John Gunnels, Greg Morrow, James Overfelt, and Robert A. van de Geijn, *Block Parallel implementation of BLAS: General techniques for level 3 BLAS*, in Concurrency: Practice and Experience, to appear.

[6] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff, *A set of level 3 Basic Linear Algebra Subprograms*, *ACM TOMS*, 16(1) (1990), pp. 1–17.

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

[8] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra, *MPI: The Complete Reference*, The MIT Press, 1996.

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

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

[11] Y.-J. J. Wu, P. A. Alpatov, C. Bischof, and R. A. van de Geijn, *A parallel implementation of symmetric band reduction using PLAPACK*, PRISM Working Note 35, in Proceedings of Scalable Parallel Library Conference, Mississippi State University, 1996.