PLAPACK: Parallel Linear Algebra Libraries Design Overview *

Philip Alpatov † Greg Baker † Carter Edwards † John Gunnels †
Greg Morrow † James Overfelt † Robert van de Geijn †‡

An Extended Abstract Submitted to SC97

Corresponding Author:

Robert van de Geijn
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712
(512) 471-9720 (office)
(512) 471-8885 (fax)
rvdg@cs.utexas.edu

Abstract

Over the last twenty years, dense linear algebra libraries have gone through three generations of public domain general purpose packages. In the seventies, the first generation of packages were EISPACK and LINPACK, which implemented a broad spectrum of algorithms for solving dense linear eigenproblems and dense linear systems. In the late eighties, the second generation package called LAPACK was developed. This package attains high performance in a portable fashion while also improving upon the functionality and robustness of LINPACK and EISPACK. Finally, Since the early nineties, an effort to port LAPACK to distributed memory networks of computers has been underway as part of the ScaLAPACK project.

PLAPACK is a maturing fourth generation package which uses a new, more application-centric, view of vector and matrix distribution, *Physically Based Matrix Distribution*. It also uses an "MPI-like" programming interface that hides distribution and indexing details in *opaque objects*, provides a natural layering in the library, and provides a straight-forward application interface. In this paper, we give an overview of the design of PLAPACK.

*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

‡Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712, phone: (512) 471-9720, fax: (512) 471-8885, rvdg@cs.utexas.edu

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 complexity of implementation. While the algorithms typically can be described without filling up more than half a chalk board, resulting codes (sequential or parallel) require 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 [2], the PETSc library at Argonne National Laboratory [1], and the Message-Passing Interface [11].

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 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 [3].

In this paper, we review the different programming styles for implementing dense linear algebra algorithms and give an overview of all components of the PLAPACK infrastructure.

2 Example: Cholesky Factorization

In this section, we illustrate the use of various components of the PLAPACK infrastructure by showing its use for a simple problem, the Cholesky factorization. We use this example to also contrast the approach that PLAPACK uses to code such algorithms with more traditional approaches used by LAPACK and ScaLAPACK.

The Cholesky factorization of a square matrix A is given by

$$A \rightarrow LL^T$$

where A is symmetric positive definite and 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 $b \times b$ ($b << n$). The $\star$ indicates the symmetric part of A. Now,

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

This in turn yields the equations

$$A_{11} \rightarrow L_{11}L_{11}^T$$

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

The above motivation yields an algorithm as it might appear on a chalk board in a classroom, in Figure 1 (a). Since A is symmetric, only the lower triangular part of the matrix is stored and only the lower triangular part is updated.

2.1 LAPACK-style implementation

LAPACK implements algorithms like the Cholesky factorization using Level-1, 2, and 3 Basic Linear Algebra Subprograms. In Fig. 1 (b), we present code for the primary loop of the Cholesky factorization in slightly simplified form. The call to `DPOTF2` executes the Cholesky factorization of the smaller matrix A_{11}, using a routine implemented with calls to Level-2 BLAS. The call to `TRSM` performs the *triangular solve with multiple right-hand-sides* $A_{21} \leftarrow A_{21}L_{11}^{-T}$. It is important to note that all indexing is derived from carefully tracking the sizes of all sub-matrices, and where they start.

2.2 ScaLAPACK implementation

The ScaLAPACK Home Page (`http://www.netlib.org/scalapack/scalapack_home.html`) states

> The ScaLAPACK (for Scalable-LAPACK) library includes a subset of LAPACK routines redesigned for distributed memory MIMD parallel computers. It is currently written in a Single-Program-Multiple-Data style using explicit message passing for interprocessor communication. It assumes matrices are laid out in a two-dimensional block cyclic decomposition.
>
> Like LAPACK, the ScaLAPACK routines are based on block-partitioned algorithms in order to minimize the frequency of data movement between different levels of the memory hierarchy. (For such machines, the memory hierarchy includes the off-processor memory of other processors, in addition to the hierarchy of registers, cache, and local memory on each processor.) The fundamental building blocks of the ScaLAPACK library are distributed memory versions (PBLAS) of the Level 1, 2 and 3 BLAS, and a set of Basic Linear Algebra Communication Subprograms (BLACS) for communication tasks that arise frequently in parallel linear algebra computations. In the ScaLAPACK routines, all interprocessor communication occurs within the PBLAS and the BLACS. One of the design goals of ScaLAPACK was to have the ScaLAPACK routines resemble their LAPACK equivalents as much as possible.

This results in the code segment for the Cholesky factorization given in Fig. 1 (c) [1]. Generally, names are routines are changed by adding a "`P`" before the LAPACK name, and parameter lists are changed by taking any occurrence of "`A(I,J), LDA`" and translating it to "`A, I, J, DESCA`". `DESCA` holds the information that describes the distribution of the matrix to processors.

2.3 PLAPACK-style implementation

Examination of the algorithm given in Fig. 1 (a), and the codes presented in (b) and (c) show that considerable effort is required to recognize the algorithm in the codes. Furthermore, the algorithm does not explicitly deal with sizes of sub-matrices or indexing into arrays. The natural way to express the algorithms is to partition matrices into quadrants. The PLAPACK project attempts to provide a programming interface that reflects this naturally. A sample implementation of the Cholesky factorization is given in Fig. 1 (d).

[1] At the time that this abstract was being written, this code segment was taken from the then current release of ScaLAPACK. While it differs from the now available release, it is representative of the coding style

We believe the code to be self-explanatory, once the algorithm itself is given. All that needs to be said is that all information about A and its distribution are encapsulated in an *opaque* object `a`, and that calls like `PLA_Obj_view_all` and `PLA_Obj_split_4` create *references* into the same data that `a` describes. Notice that the bulk of the computation continues to be performed in routines that mirror the traditional BLAS, except that many of the parameters have now been absorbed into the opaque objects.

2.4 Find the Error in the Code

To illustrate the benefits of the PLAPACK coding style, we have introduced an error in each of the codes in Figure 1 (b)–(d). We invite the reader to find the errors!

3 Building a Parallel (Dense) Linear Algebra Infrastructure

In Figure 2, we show a schematic of the layering of PLAPACK. In this section, we discuss briefly all components of this infrastructure.

3.1 To C++ or not to C++

These days, a primary issue that must be resolved upon starting a new software project is the choice of the language. The fact that PLAPACK naturally uses an object-based paradigm of programming seems to favor the use of C++. Unfortunately, a number of reasons dictate that a more traditional language be used. These include the fact that it must be possible to layer applications written in C or FORTRAN on top of the library, the availability of compilers, especially ones that implement a standardized version of the language chosen. It is for this reason that we chose the C programming language, creating an "MPI-like" interface.

3.2 Machine/distribution dependent layer

When designing a new library infrastructure, it is important to ensure portability by either limiting the components that must be customized, or by using only standardized components and/or languages. PLAPACK achieves this by using standardized components, namely the Message-Passing Interface (MPI) for communication, standard memory management routines provided by the C programming language (`malloc/calloc`), and the Basic Linear Algebra Subprograms (BLAS) which are generally optimized by the vendors. To this layer, we also add Cartesian matrix distributions, since they provide the most general distribution of matrices commonly used for parallel dense linear algebra algorithm implementation.

3.3 Machine/distribution independent layer

PLAPACK provides a number of interfaces to the machine dependent layer to achieve machine independence. Below we give a brief description of each.

PLAPACK-MPI interface: PLAPACK relies heavily on collective communications like scatter (`MPI_Scatter`), gather (`MPI_Gather`), collect (`MPI_Allgather`), broadcast (`MPI_Bcast`), and others. However, we have created an intermediate layer that can be used to pick a particular implementation of such an operation. Reasons include the fact that often communication and computation can be orchestrated to flow in the same direction, allowing for transparent overlapping of computation and communication. In addition, often vendor implementations of collective communication are either inefficient or unpredictable in their cost. Predictability is important when trying to hybridize implementations for different situations. Such hybridizations are sometimes called *poly-algorithms*. Our own experience implementing

the Interprocessor Collective Communication Library (InterCom) [10] will allow us in the future more flexibility in minimizing the overhead from communication within PLAPACK.

PLAPACK-memory management interface: PLAPACK heavily uses dynamic memory allocation to create space for storing data. By creating this extra layer, we provide a means for customizing memory management, including the possibility of allowing the user to provide all space used by PLAPACK.

Physically Based Matrix Distribution (PBMD) and Templates: Fundamental to PLAPACK is the approach taken to describe Cartesian matrix distributions. In particular, PLAPACK recognizes the (most?) important role that vectors play in applications and thus all distribution of matrices and vectors starts with distribution of vectors. The details of the distribution are hidden by describing the generic distribution of imaginary vectors and matrices (the template) and indicating how actual matrices and vectors are aligned to the template.

PLAPACK-BLAS interface: Locally on each processing node, computation is generally performed by the BLAS, which have a FORTRAN compatible calling sequence. Since the interface between C and FORTRAN is not standardized, a PLAPACK-BLAS interface is required to hide these platform specific differences.

3.4 PLAPACK abstraction layer

This layer of PLAPACK provides the abstraction that allows one to remove oneself from details like indexing, communication, and local computation.

Linear Algebra Objects and their manipulation: As illustrated in Section 2, all information that describes a linear algebra object like a vector or matrix is encapsulated in an opaque object. This component of PLAPACK allows one to create, initialize, and destroy such objects. In addition, it provides an abstraction which allows one to transparently index into a sub-matrix or vector. Finally, it provides a mechanism for describing duplication and data to be consolidated.

Copy/Reduce: Duplication and consolidation of data: Using the PLAPACK linear algebra objects one describes how data is (to be) distributed and/or duplicated. Given this ability, communication is now hidden by instead copying from one object to another or reducing the contents of an object that is duplicated and may contain inconsistent data or partial contributions.

PLAPACK Local BLAS:

Since all information about matrices and vectors is hidden in the linear algebra objects, a call to a BLAS routine on a given node requires extraction of that information. Rather than exposing this, PLAPACK provides routines called the PLAPACK Local BLAS which extract the information and subsequently call locally the correct sequential BLAS routine.

3.5 Library layer

The primary intent for the PLAPACK infrastructure is to provide the building blocks for creating higher level libraries. Thus, the library layer for PLAPACK consists of global (parallel) basic linear algebra subprograms and higher level routines for solving linear systems and algebraic eigenvalue problems.

PLAPACK Global BLAS: The primary building blocks provided by PLAPACK are the global (parallel) versions of the BLAS. These allow dense linear algebra algorithms to be implemented quickly without exposing parallelism in any form, as was illustrated for the right-looking Cholesky factorization in

Section 2, achieving without much effort a respectable percentage of peak performance on a given architecture.

PLAPACK higher level linear algebra routines: As illustrated in Section 2, higher level algorithms can be easily implemented using the global BLAS. However, to get better performance it is often desirable to implement these higher level algorithms directly using the abstraction level of PLAPACK, explaining the dashed lines in Fig. 2. Development of such implementations can often be attained by incrementally replacing calls to global BLAS with calls that explicitly expose parallelism by using objects that are duplicated in nature. For details, we refer the reader to [12].

3.6 PLAPACK Application Interface

A highlight of the PLAPACK infracture is the inclusion of a set of routines that allow an application to build matrices and vectors in an application friendly manner. This interface is called the Application Program Interface (API), where we use the word *application* to refer to the application that uses the library, rather than the more usual computer science meaning[2].

To understand how the API is structured, one must first realize how a typical engineering application builds a matrix. Rather than being "passed down from God" as Ivo Babuska would say, the matrix is often generated by adding sub-matrices to a matrix in an overlapped fashion. Each sub-matrix can often be generated on one node, with many being generated in parallel. The process of submitting a sub-matrix to the global matrix then requires communication.

Managed Message-Passing Interface: To simplify the process of adding a sub-matrix to a global matrix, it is desirable to use a *one-sided* communication protocall, rather than the usual message-passing currently provided by MPI. However, for portability reasons, such a mechanism itself must be implemented using only standard MPI. The Managed Message-Passing Interface (MMPI) developed by Carter Edwards at the University of Texas provides such a mechanism [7].

PLAPACK Application Program Interface (API): A typical PLAPACK call for submitting a sub-matrix to a global matrix is given by

```
PLA_API_axpy_matrix_to_global ( m, n, *alpha, *a_local, lda, a_global, displ_row, displ_col )
```

This call indicates that α times the $m \times n$ matrix at location `a_local` on this node is to be added to the equally sized sub-matrix of global matrix described by object `a_global`. This sub-matrix of the global matrix has its upper-left-hand element at location (`displ_row, displ_col`) of the global matrix. This call is only performed on the node that is submitting the matrix, making it a one-sided call implemented using MMPI.

4 Performance

In Figure 3, we show the performance of right-looking Cholesky factorization implemented using PLAPACK. The timings reported are for an implementation that is coded at a slightly lower level of PLAPACK than described in Section 2. The performance numbers were obtained on the Jet Propulsion Laboratory's Cray T3D. For reference, matrix-matrix multiplication on a single node attains around 105-120 MFLOPS/sec on that architecture.

Parallelizing Cholesky factorization as a *left-looking* algorithm is actually more natural, and yields better performance. While not the subject of this paper, we also report performance for that variant.

[2] The term application program interface in computer science would generally be used for the entire PLAPACK infrastructure.

To showcase performance of PLAPACK on various parallel architectures, we also include a performance graph for the PLAPACK matrix-matrix multiplication kernel, in Figures 4 and 5. Reference performance levels attainable on single processors for these architectures are given by

Intel Paragon	46 MFLOPS/sec
Cray T3D	120 MFLOPS/sec
IBM SP-2	210 MFLOPS/sec
Convex Exemplar S-series	510 MFLOPS/sec

5 Conclusion

While not yet possessing the full functionality of previous generation parallel linear algebra libraries, PLAPACK provides an infrastructure upon which parallel linear algebra routines may be quickly customized and optimized for the specific user application. This is in contrast to the customization of user applications required to effectively use previous generation parallel linear algebra libraries.

Several efforts currently use PLAPACK as an essential component of their high level applications. All of these efforts have reported performance which is competitive with reported performance of more established libraries with a significant reduction in application development time and lines of code. Examples include the use of PLAPACK to implement a parallel reduction to banded form algorithm as part of the PRISM project [14], its use in solving an interface problem as part of a parallel multifrontal solver for an h-p adaptive finite element method [8], and its use in solving problems in satellite geodesy [5].

Acknowledgments

We would like to thank Dr. Yuan-Jye (Jason) Wu for agreeing to be a one-man alpha-release test site for PLAPACK. By implementing a complex algorithm (the reduction of a square dense matrix to banded form [14]) using PLAPACK, Jason exercised many components of the infrastructure, thereby revealing many initial bugs. We also appreciate the encouragement received from Dr. Chris Bischof, Dr. Steven Huss-Lederman, Prof. Xiaobai Sun, and others involved in the PRISM project. We gratefully acknowledge George Fann at PNNL for his support of the PLAPACK project and for his time collecting the IBM SP-2 performance numbers presented. Similarly, we are indebted to Dr. Greg Astfalk for assembling the numbers for the Convex Exemplar.

We gratefully acknowledge access provided by the following parallel computing 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.) Cray Research, a Silicon Graphics Company. The Cray T3D at the Jet Propulsion Laboratory. The Convex Division of the Hewlett-Packard Company in Richardson, Texas.

Additional information

For additional information, visit the PLAPACK web site:

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

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] G. Baker, *Application of Parallel Processing to Selected Problems in Satellite Geodesy*, Ph.D. dissertation, Department of Aerospace Engineering, The University of Texas at Austin, in preparation.

[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] H. Carter Edwards, *MMPI: Asynchronous Message Management for the Message-Passing Interface*, TICAM Report 96-44, The University of Texas at Austin, 1996.

[8] H. Carter Edwards, *A Parallel Infrastructure for Scalable Adaptive Finite Element Methods and its application to Least Squares C-infinity Collocation*, Ph.D. dissertation, Computational and Applied Mathematics Program, The University of Texas at Austin, 1997.

[9] 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.

[10] P. Mitra, D. Payne, L. Shuler, R. van de Geijn, and J. Watts, "Fast Collective Communication Libraries, Please," Proceedings of the Intel Supercomputing Users' Group Meeting 1995.

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

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

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

[14] 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.

let $A_{\text{cur}} = A$
do until A_{cur} is 0×0
Partition
$A_{\text{cur}} = \left(\begin{array}{c|c} A_{11} & \star \\ \hline A_{21} & A_{22} \end{array}\right)$
$A_{11} \leftarrow L_{11} = \text{Chol factor of } A_{11}$
$A_{21} \leftarrow L_{21} = A_{21} L_{11}^{-T}$
$A_{22} \leftarrow A_{22} - L_{21} L_{21}^{T}$
continue with $A_{\text{cur}} = A_{22}$

(a) Level-3 BLAS algorithm

```
 DO J = 1, N, NB
   JB = MIN( N-J+1, NB )

   CALL DPOTF2( 'Lower', JB, A(J,J), LDA, INFO )

   IF ( J-JB .LE. N )
     CALL DTRSM( 'Right', 'Lower', 'Trans',
$          'Nonunit', N-J-JB+1, JB, ONE,
$          A( J, J ), LDA, A( J+JB, J ), LDA )

     CALL DSYRK( 'Lower', 'No transpose', N-J-JB+1,
$          JB, -ONE, A( J+JB, J ), LDA,
$          A( J, J ), LDA )
   END IF
 END DO
```

(b) Level-3 BLAS implementation

```
 DO 20 J = JN+1, JA+N-1, DESCA( NB_ )
   JB = MIN( N-J+JA, DESCA( NB_ ) )
   I = IA + J - JA

   CALL PDPOTF2( 'Lower', JB, A, I, J, DESCA, INFO )

   IF( J-JA+JB+1.LE.N ) THEN

     CALL PDTRSM( 'Right', 'Lower', 'Transpose',
$          'Non-Unit', N-J-JB+JA, JB, ONE,
$          A, I, J, DESCA, A, I+JB, J, DESCA )

     CALL PDSYRK( 'Lower', 'No Transpose', N-J-JB+JA,
$          JB, -ONE, A, I+JB, J, DESCA, ONE, A,
$          I, J, DESCA )
   END IF
 END DO
```

(c) ScaLAPACK Level-3 PBLAS implementation

```
PLA_Obj_view_all( a, &acur );
while ( TRUE ){
   PLA_Obj_global_length( acur, &size );
   if ( 0 == ( size = min( size, nb ) ) )
      break;
   PLA_Obj_split_4( acur, size, size,
                                &a11, &a12,
                                &a21, &acur );
   Chol2( 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,
             min_one, a21, a );
}
```

(d) PLAPACK Level-3 BLAS implementation

Figure 1: Comparison of various methods for coding right-looking Cholesky factorization, using matrix-matrix operations.

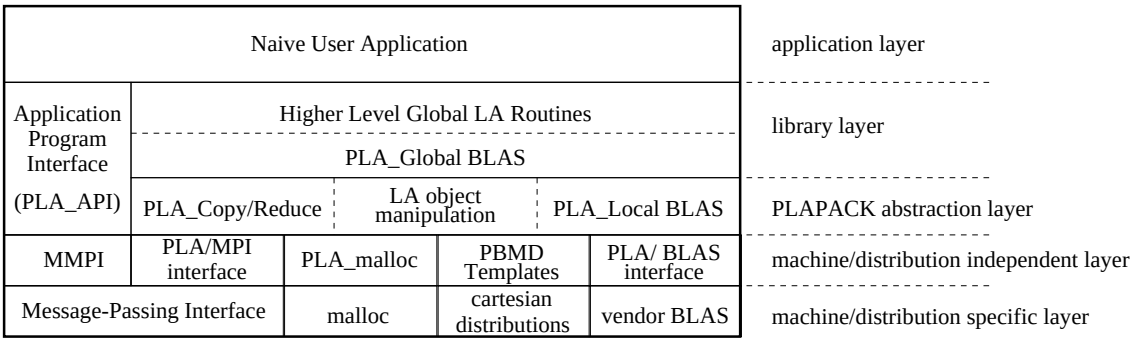

Figure 2: Layering of the PLAPACK infrastructure

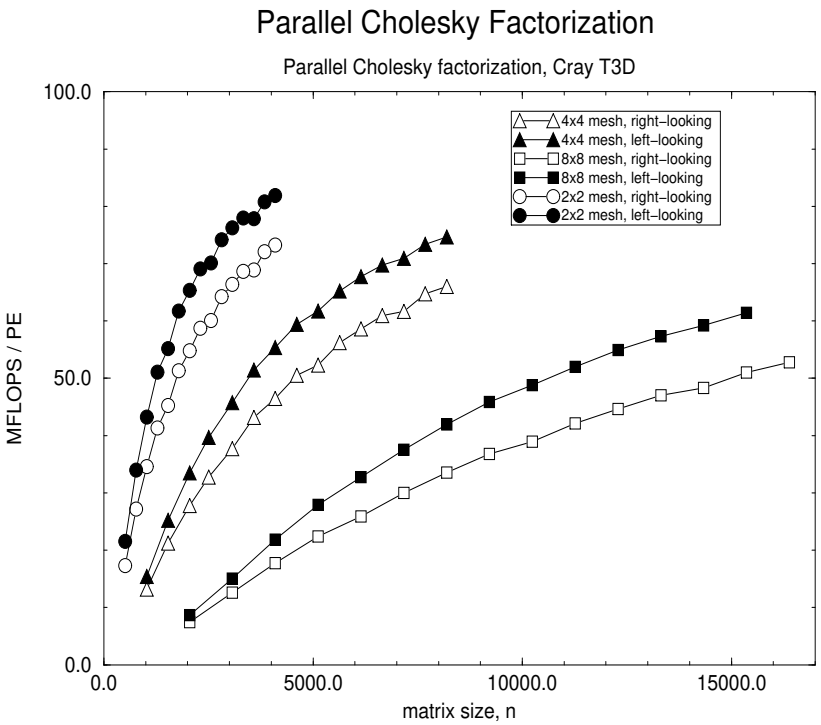

Figure 3: Performance of Cholesky factorization using PLAPACK

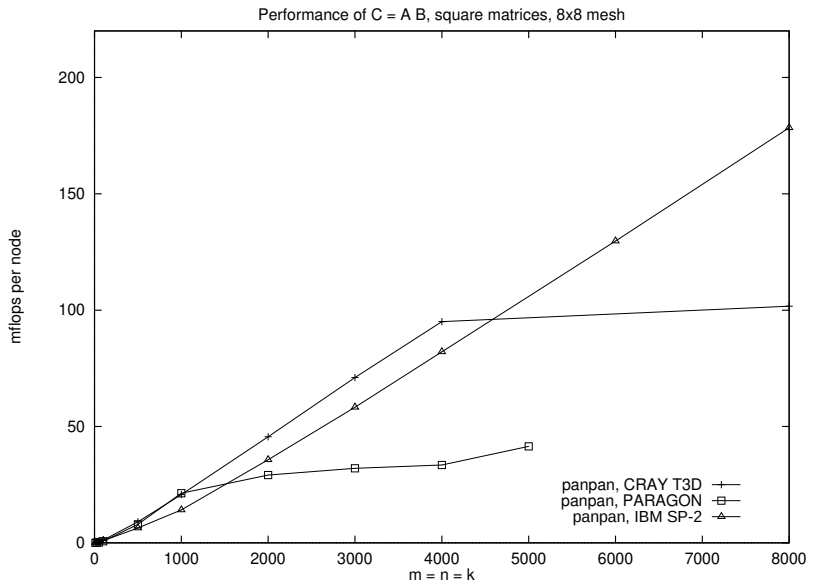

Figure 4: Performance of $C = \alpha AB + \beta C$ on 64 processor configurations of the Intel Paragon with GP nodes, the Cray T3D, and the IBM SP-2.

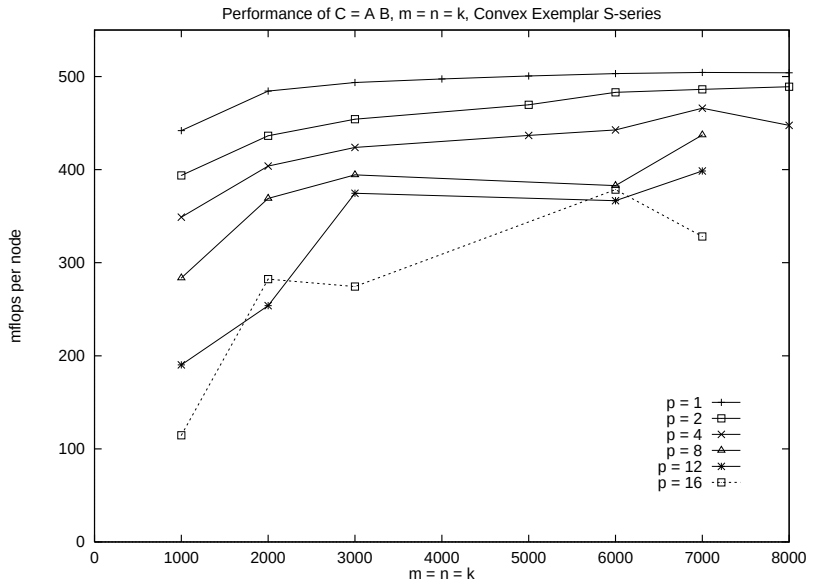

Figure 5: Performance of $C = \alpha AB + \beta C$ for the panel-panel variant on the Convex Exemplar S-series. The matrices are taken to be square and their dimensions are varied along the x- axis. The different curve reflect performance when the number of nodes, p, is chan ged.