Application Interface to Parallel Dense Matrix Libraries: Just let me solve my problem!

H. Carter Edwards
Sandia National Laboratories
P.O. Box 5800 / MS 0382
Albuquerque, NM 87185
hcedwar@sandia.gov

Robert A. van de Geijn
Department of Computer Sciences
The University of Texas at Austin
1 University Station C0500
Austin, TX 78712
rvdg@cs.utexas.edu

February 20, 2006

PLAPACK Working Note #13
FLAME Working Note #18

Abstract

We focus on how applications that lead to large dense linear systems naturally build matrices. This allows us explain why traditional interfaces to dense linear algebra libraries for distributed memory architectures, which evolved from sequential linear algebra libraries, inherently do not support applications well. We review the application interface that has been supported by the Parallel Linear Algebra Package (PLAPACK) for almost a decade, which appears to support applications better. The lesson learned is that an application-centric interface can be easily defined, deminishing the obstacles that exist when using large distributed memory architectures.

1 Introduction

Application-centric interfaces to dense linear algebra libraries can greatly simplify the task of porting a typical application that utilizes such libraries to distributed memory architectures. The classic doctrine has been that if your library has the right functionality and is fast, it will be used. A consequence of the doctrine has been that the application interface to the library has typically been library-centric. For sequential dense linear algebra libraries like the Linear Algebra Package (LAPACK), this doctrine led to obvious success: LAPACK is undoubtedly the most widely used library for this problem domain. That this doctrine does not apply in general is obvious from the frustration voiced by users of ScaLAPACK, the extension of LAPACK for distributed memory architectures [6, 2]. It is by abandoning this doctrine and focusing on application-centric interfaces to libraries that these shortcomings can be avoided without sacrificing performance or functionality.

The primary contribution of this paper is to prominently register the importance of application-centric interfaces. Such interfaces greatly reduce the effort required to parallelize complex applications and should therefore be a primary concern of library developers. We demonstrate that they are easy to define and straightforward to implement for the problem domain targeted by ScaLAPACK.

In Section 2 we analyze a prototypical application that gives rise to a large dense linear system and propose an abstraction for parallelizing the generation of the matrix. In Section 3 we show how the application interface of the Parallel Linear Algebra Package (PLAPACK) [1, 3, 18, 17] supports the abstractions needed to parallelize the prototypical application. Related work is discussed in Section 4. Conclusions can be found in the final section.

2 Analysis of A Prototypical Application

Practical applications solved using the Boundary Element Method (BEM) often lead to very large dense linear systems[1]. The idea there is that by placing the discretization on the boundary of a three-dimensional object, the degrees of freedom are restricted to a two-dimensional surface. In contrast, Finite Element Methods (FEM) set degrees of freedom throughout the three dimensional object. The resulting reduction of degrees of freedom comes at a price: BEM elements have all-to-all interaction, each element-element interaction defines a submatrix, all elements of that submatrix are generated as unit, and the resulting matrix is typically complex valued [8, 9, 12].

In Fig. 1 we show for a discretization with only two elements how the formation of the global matrix is inherently additive. In the case of a 2D discretization (the surface of a 3D object), more interfaces between elements occur and the mapping to the global matrix is somewhat more complex, but the same principles apply. When hp-adaptive discretization is used and/or the application involves multi-physics, the number of degrees of freedom per element can be nonuniform, leading to local matrices with nonuniform dimensions. In Fig. 3 we show how for a discretization with N elements the linear system is computed via a double loop, each over all elements.

The computation of the local submatrix inherently computes all elements of that submatrix simultaneously, rather than one element at a time. Thus, computing the global matrix one element at a time is not practical.

In Fig. 4 we show how conceptually the computation of the global matrix can be parallelized. Here the `I_compute` function returns *true* if and only if the node with index `me` is to compute the row of blocks $\{A_{i,0}, \ldots, A_{i,N-1}\}$. Since there is relatively little data associated with the discretization and the physical parameters, that data can be assumed to have been duplicated to all nodes.

There are two requirements for the approach in Fig. 4:

- Communication must occur in order to add each contributed submatrix $A^{(i,j)}$ into the global, distributed matrix A.
- Ideally this communication is transparent to the application such that the *application* is not required to explicitly receive the contributed submatrix (or any entries of the submatrix) on the nodes that "own" those portions of the global, distributed matrix.

We note that another important application that similarly generates contributions to a global matrix is a sparse direct solver, where the matrix that corresponds to the interface problem is filled with contributions from disjoint interior regions [11].

[1]New methods are increasingly being used to solve such problems iteratively. These methods use the Fast Multipole Method (FMM) or other fast summation method to accelerate the matrix-vector multiply. Nonetheless, many applications resort to solving such problems by forming a dense linear system instead. Regardless, our example demonstrates how applications often interface to dense solvers.

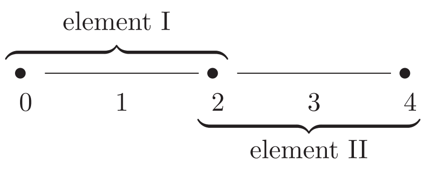

Figure 1: A simple discretization.

$$
\overbrace{\left(\begin{array}{c|c|c|c|c}
A_{00}^{(\mathrm{I,I})} & A_{01}^{(\mathrm{I,I})} & A_{02}^{(\mathrm{I,I})} & 0 & 0 \\ \hline
A_{10}^{(\mathrm{I,I})} & A_{11}^{(\mathrm{I,I})} & A_{12}^{(\mathrm{I,I})} & 0 & 0 \\ \hline
A_{20}^{(\mathrm{I,I})} & A_{21}^{(\mathrm{I,I})} & A_{22}^{(\mathrm{I,I})} & 0 & 0 \\ \hline
0 & 0 & 0 & 0 & 0 \\ \hline
0 & 0 & 0 & 0 & 0
\end{array}\right)}^{\text{Coupling of I with I } (A^{(\mathrm{I,I})})}
+
\overbrace{\left(\begin{array}{c|c|c|c|c}
0 & 0 & 0 & 0 & 0 \\ \hline
0 & 0 & 0 & 0 & 0 \\ \hline
0 & 0 & A_{22}^{(\mathrm{II,II})} & A_{23}^{(\mathrm{II,II})} & A_{24}^{(\mathrm{II,II})} \\ \hline
0 & 0 & A_{32}^{(\mathrm{II,II})} & A_{33}^{(\mathrm{II,II})} & A_{34}^{(\mathrm{II,II})} \\ \hline
0 & 0 & A_{42}^{(\mathrm{II,II})} & A_{43}^{(\mathrm{II,II})} & A_{44}^{(\mathrm{II,II})}
\end{array}\right)}^{\text{Coupling of II with II } (A^{(\mathrm{II,II})})}
$$

$$
+
\overbrace{\left(\begin{array}{c|c|c|c|c}
0 & 0 & A_{02}^{(\mathrm{I,II})} & A_{03}^{(\mathrm{I,II})} & A_{04}^{(\mathrm{I,II})} \\ \hline
0 & 0 & A_{12}^{(\mathrm{I,II})} & A_{13}^{(\mathrm{I,II})} & A_{14}^{(\mathrm{I,II})} \\ \hline
0 & 0 & A_{22}^{(\mathrm{I,II})} & A_{23}^{(\mathrm{I,II})} & A_{24}^{(\mathrm{I,II})} \\ \hline
0 & 0 & 0 & 0 & 0 \\ \hline
0 & 0 & 0 & 0 & 0
\end{array}\right)}^{\text{Coupling of I with II } (A^{(\mathrm{I,II})})}
+
\overbrace{\left(\begin{array}{c|c|c|c|c}
0 & 0 & 0 & 0 & 0 \\ \hline
0 & 0 & 0 & 0 & 0 \\ \hline
A_{20}^{(\mathrm{II,I})} & A_{21}^{(\mathrm{II,I})} & A_{22}^{(\mathrm{II,I})} & 0 & 0 \\ \hline
A_{30}^{(\mathrm{II,I})} & A_{31}^{(\mathrm{II,I})} & A_{32}^{(\mathrm{II,I})} & 0 & 0 \\ \hline
A_{40}^{(\mathrm{II,I})} & A_{41}^{(\mathrm{II,I})} & A_{42}^{(\mathrm{II,I})} & 0 & 0
\end{array}\right)}^{\text{Coupling of II with I } (A^{(\mathrm{II,I})})}
$$

$$
= \left(\begin{array}{c|c|c|c|c}
A_{00}^{(\mathrm{I,I})} & A_{01}^{(\mathrm{I,I})} & A_{02}^{(\mathrm{I,I})} + A_{02}^{(\mathrm{I,II})} & A_{03}^{(\mathrm{I,II})} & A_{04}^{(\mathrm{I,II})} \\ \hline
A_{10}^{(\mathrm{I,I})} & A_{11}^{(\mathrm{I,I})} & A_{12}^{(\mathrm{I,I})} + A_{12}^{(\mathrm{I,II})} & A_{13}^{(\mathrm{I,II})} & A_{14}^{(\mathrm{I,II})} \\ \hline
A_{20}^{(\mathrm{I,I})} + A_{20}^{(\mathrm{II,I})} & A_{21}^{(\mathrm{I,I})} + A_{21}^{(\mathrm{II,I})} & \begin{array}{c} A_{22}^{(\mathrm{I,I})} + A_{22}^{(\mathrm{II,II})} \\ + A_{22}^{(\mathrm{I,II})} + A_{22}^{(\mathrm{II,I})} \end{array} & A_{23}^{(\mathrm{III,I})} + A_{23}^{(\mathrm{II,I})} & A_{24}^{(\mathrm{II,II})} + A_{24}^{(\mathrm{II,I})} \\ \hline
A_{30}^{(\mathrm{II,I})} & A_{31}^{(\mathrm{II,I})} & A_{32}^{(\mathrm{II,II})} + A_{32}^{(\mathrm{II,I})} & A_{33}^{(\mathrm{II,II})} & A_{34}^{(\mathrm{II,II})} \\ \hline
A_{40}^{(\mathrm{II,I})} & A_{41}^{(\mathrm{II,I})} & A_{42}^{(\mathrm{II,II})} + A_{42}^{(\mathrm{II,I})} & A_{43}^{(\mathrm{II,II})} & A_{44}^{(\mathrm{II,II})}
\end{array}\right)
$$

Figure 2: Contributions from the coupling between the two elements yield the global stiffness matrix.

```
$A = 0$
**for** $i = 0, ..., N-1$
    **for** $j = 0, ..., N-1$
        **Compute** coupling matrix $A^{(i,j)}$
        **Add** $A^{(i,j)}$ into $A$
    **endfor**
**endfor**
```

Figure 3: Simple look that computes all element-element interactions.

3 An Exemplar Application Interface

We now discuss an interface for filling matrices and vectors that has been successfully employed by PLAPACK since its inception in 1997. We assume that the reader is familiar with MPI and how it uses object-based programming [16], as well as the C programming language.

```
$A = 0$
**for** $i = 0, ..., N-1$
    **if** I_compute($i$) == me **then**
        **for** $j = 0, ..., N-1$
            **Compute** coupling matrix $A^{(i,j)}$
            **Add** $A^{(i,j)}$ into $A$ (requires communication)
        **endfor**
    **endif**
**endfor**
```

Figure 4: Simple (but effective) parallelization of the computation in Fig. 3.

3.1 A simple example

We employ a simple example: Partition matrix A by columns, $A = \left(a_0 \mid \cdots \mid a_{n-1} \right)$. Given p nodes, the code in Fig. 5 computes columns a_j on node (j mod p) and submits it for addition to a global matrix A. Details of the interface are

Line 4: Global matrix A and related information (like its distribution) are encapsulated in the linear algebra object `A`, which is passed to the subroutine.

Line 15: Initialize A to the zero matrix.

Lines 17 & 37: These begin/end functions initiate and finalize the "behind the scenes" communication mechanism that allows each node to perform independent "submatrix add into global matrix" operations. These are global synchronous function calls.

Lines 19 & 35: These open/close functions first open the global matrix `A` to local, independent submission (or extraction) of submatrices, and then resolves all of the submissions into (or extractions from) `A`. These are global synchronous function calls.

Line 21: Create local space for a single column, `local_a_j`.

Lines 23–31: Loop that fills columns and submits them for addition to the global matrix. Column j is created on node (j mod p) and submitted to the global matrix.

Line 29: Submit column j for addition to the global matrix. Here `n, 1` indicates the dimension of the matrix being submitted (in this case a column); `d_one` indicates that $1.0 \times$ `local_a_j` is to be added to the jth column of matrix A; `local_a_j` is the address, locally, where the matrix being submitted resides; `n` is the leading dimension of the array in which the matrix being submitted resides; `A` is the descriptor of the global matrix; `0, j` indicates that a_j is to be added to the submatrix of A that has its top-left element at index $(0, j)$ in matrix A. This call is only made on the node that is submitting data. It is *not* a synchronous call.

Extracting data from a global matrix can be similarly accommodated.

The "add into" calls like the one on Line 29 merely *submit* contributions. These contributions are **not** guaranteed to be resolved until the close function (Line 35) is called for the global, distributed object `A`. The "behind the scenes" communication mechanism for this submit & resolve strategy could have a variety of implementations, depending upon the underlying communication library (MPI-1, MPI-2, OpenMP, etc.) and performance considerations. Typically performance focuses on minimizing the time-to-fill, i.e. minimize the time between the begin & end operations (Lines 17 & 37) inclusively. Other performance

```
#include "mpi.h"
#include "PLA.h"

void create_problem( PLA_Obj A )
{
  int n, me, nprocs, i, j;
  double *local_a_j, d_one = 1.0;

  PLA_Obj_global_length( A, &n );     /* Extract matrix dimension */

 /* Extract this node's rank and the total number of nodes */
  MPI_Comm_rank( MPI_COMM_WORLD, &me );
  MPI_Comm_size( MPI_COMM_WORLD, &nprocs );

  PLA_Obj_set_to_zero ( A );     /* Initialize A = 0 */

  PLA_API_begin();     /* Start of critical section for filling */

    PLA_Obj_API_open(A);       /* Open A for contributions */

    local_a_j = (double *) malloc( sizeof( double ) * n );

    for ( j=0; j<n; j++ ) {
      if ( j%nprocs == me ){   /* if ( j mod nprocs ) == me fill column */
        for ( i=0; i<n; i++ )
          local_a_j[i] = ( double ) i + j*0.001;  /* A(i,j) = i + j*0.001 */

        /* A( 0:n-1,: ) = A( 0:n-1,: ) + local_a_j; */
        PLA_API_axpy_matrix_to_global( n, 1, &d_one, local_a_j, n, A, 0, j );
      }
    }

    free( local_a_j );

    PLA_Obj_API_close(A);      /* Close A for contributions */

  PLA_API_end();       /* End of critical section for filling */
}
```

Figure 5: A sample subroutine for filling a matrix A.

considerations could include memory overhead, or more significantly whether the overall fill operation should be deterministic or is allowed to be nondeterministic.

3.2 Determinism: a performance trade-off

A deterministic fill operation would guarantee that **if** the global matrix `A` were identically distributed, an identical set of local submatrix contributions were made, and the application code is executed on an identical computer **then** the resulting data in `A` will be identical. Consider the two following simple summations:

$$a = \sum_{i=0}^{N-1} x_i \quad \text{versus} \quad b = \sum_{i=N-1}^{0} x_i \quad \text{(reverse order)}.$$

For finite precision arithmetic a cannot be guaranteed to be equal to b due to the roundoff resulting from a different ordering of contributions. Similarly, in order to guarantee that a fill operation produces identical results (is deterministic), the implementation must guarantee that the submatrix contributions are summed in an identical order. Given the non-deterministic nature of a typical communication mechanism (e.g. MPI), requiring the parallel fill operation to be deterministic would cause the implementation to use more time and/or memory. For PLAPACK this performance trade-off favored speed and as such does not require the parallel fill operation to be deterministic.

3.3 Implementation history

The initial implementation of the PLAPACK Application Interface (1997) utilized the Managed Message-Passing Interface (MMPI) [10], a package that layers one-sided communication on top of the MPI-1 standard. This implementation was replaced in 1999 in order to minimize the number of messages (communication startup cost) while simultaneously ensuring that MPI would not run out of buffer space. The current implementation accumulates submatrix contributions on the local node and then generates a single MPI message from the contributing node to the “owning” node that contains data from multiple contributions. Future implementation improvements could include the use of pthreads or the one sided communication mechanisms in the MPI-2 standard.

3.4 Performance Issues

Given that computation on dense matrices typically involves $O(n^3)$ operations and that the communication of data to where it belongs in a global matrix inherently involves $O(n^2)$ data, the cost of the communication is invariably negligible once the matrix becomes large. What *is* important is that the work associated with the computation of matrix entries is load-balanced among the nodes, something that is conveniently facilitated by the demonstrated interface. We do not show performance results.

4 Related Work

The need for an application-centric interface to parallel distributed linear algebra libraries was recognized by the authors over a decade ago [11] and implemented in PLAPACK. In the same time-frame, other research & development projects also recognized the importance of, and have acted upon, this need. To mention a few:

- Global Arrays Toolkit [15] at Pacific Northwest Laboratory,

- Portable, Extensible Toolkit for Scientific Computation (PETSc) [4, 5] at Argonne National Laboratory, and
- Finite Element Interface (FEI) to solvers [7] at Sandia National Laboratories,

All of these projects provide application-centric / application-friendly interfaces for filling parallel distributed matrices and similarly for extracting submatrices.

Unfortunately ScaLAPACK, currently the most widely used dense linear algebra library for distributed memory architectures, has yet to act upon the need for an application-friendly interface. ScaLAPACK employs a two-dimensional block-cyclic distribution of matrices to nodes. Applications must either build the matrix according to this distribution, requiring the application programmer to understand the intricate details of the distribution, or must build the matrix according to some other distribution, after which a redistribution routine is provided. The fundamental problem is that both the ScaLAPACK native distribution and the distributions that are supported by the redistribution routines have the property that each entry in the matrix can only reside on one node. As a result, all communication required to fill the matrix is the responsibility of the application, or, alternatively, redundant computation must be performed. This interface is entirely library-centric, which has led to considerable frustration among its users [13].

5 Conclusion

The importance of an application-friendly interface to libraries cannot be understated, especially in the setting of complex data structures for storing matrices. The PLAPACK interface demonstrates that such an interface is easily defined and implemented. The consequences of declining to address this need has been clearly demonstrated by the frustrations of the users of ScaLAPACK.

The distribution of a dense matrix to a distributed memory architecture is only one example of a complex data structure that stores matrices. Recently, the storage of dense matrices by blocks (as opposed to column-major order) has become a prominent research topic, since it is believe to better map to architectures with complex multi-level memories. The complications of filling such matrices presents challenges similar to those discussed in this paper. In [14] it is shown that an interface similar to the one proposed in this paper again yields an application-friendly solution.

Acknowledgments

PLAPACK was sponsored in part by the Intel Research Council, The Parallel Research on Invariant Subspace Methods (PRISM) project (under ARPA grant P-95006), the NASA High Performance Computing and Communications Program's Earth and Space Sciences Project (under NRA Grants NAG5-2497 and NAG5-2511), and the Environment Molecular Sciences Construction Project at Pacific Northwest National Laboratory (PNNL). We also thank the many contributors to the PLAPACK project.

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, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide - Release 2.0.* SIAM, 1994.

[3] Greg Baker, John Gunnels, Greg Morrow, Beatrice Riviere, and Robert van de Geijn. PLAPACK: High performance through high level abstraction. In *Proceedings of ICPP98*, 1998.

[4] Satish Balay, William Gropp, Lois Curfman McInnes, and Barry Smith. PETSc 2.0 users manual. Technical Report ANL-95/11, Argonne National Laboratory, Oct. 1996.

[5] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

[6] 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*, pages 120–127. IEEE Comput. Soc. Press, 1992.

[7] R. Clay, K. Mish, I. Otero, L. Taylor, and A. Williams. An annotated reference guide to the finite element interface specification version 1.0. Technical Report SAND99-8229, Sandia National Laboratories, Albuquerque, New Mexico 87185 and Livermore, California 94550, 1999.

[8] Tom Cwik, Robert van de Geijn, and Jean Patterson. The application of parallel computation to integral equation models of electromagnetic scattering. *Journal of the Optical Society of America A*, 11(4):1538–1545, April 1994.

[9] L. Demkowicz, A. Karafiat, and J.T. Oden. Solution of elastic scattering problems in linear acoustics using *h-p* boundary element method. *Comp. Meths. Appl. Mech. Engrg*, 101:251–282, 1992.

[10] H. Carter Edwards. MMPI: Asynchronous message management for the message-passing interface. TICAM Report 96-44, Texas Institute for Computational and Applied Mathematics, The University of Texas at Austin, 1996.

[11] H. Carter Edwards. *A Parallel Infrastructure for Scalable Adaptive Finite Element Methods and Its Application to Least Squares C^∞ Collocation.* PhD thesis, CAM Program, The University of Texas at Austin, 1997.

[12] Po Geng, J. Tinsley Oden, and Robert van de Geijn. Massively parallel computation for acoustical scattering problems using boundary element methods. *Journal of Sound and Vibration*, 191(1):145–165, 1996.

[13] LAPACK and ScaLAPACK survey, 2005.
`http://icl.cs.utk.edu/lapack-forum/survey/`.

[14] Tze Meng Low and Robert van de Geijn. An api for manipulating matrices stored by blocks. Technical Report CS-TR-04-14, Department of Computer Sciences, The University of Texas at Austin, May 2004. `http://www.cs.utexas.edu/users/flame/pubs/`.

[15] J. Nieplocha, R.J. Harrison, and RJ Littlefield. Global arrays: A nonuniform memory access programming model for high-performance computers. *The Journal of Supercomputing*, 10:197–220, 1996.

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

[17] Robert van de Geijn. Zen and the art of high performance parallel computing. PLAPACK Tutorial available from `http://www.cs.utexas.edu/users/plapack`.

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