CONCURRENCY: PRACTICE AND EXPERIENCE
*Concurrency: Pract. Exper.* 2000; **12**:1481–1493

# The design and implementation of the parallel out-of-core ScaLAPACK LU, QR, and Cholesky factorization routines

Eduardo D'Azevedo[1,*] and Jack Dongarra[2]

[1] *Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831–6367, U.S.A.*
[2] *Computer Science and Mathematics Division, Oak Ridge National Laboratory. Department of Computer Science, University of Tennessee, Knoxville, Tennessee 37996-1301, U.S.A.*

## SUMMARY

**This paper describes the design and implementation of three core factorization routines—LU, QR, and Cholesky—included in the out-of-core extension of ScaLAPACK. These routines allow the factorization and solution of a dense system that is too large to fit entirely in physical memory. The full matrix is stored on disk and the factorization routines transfer sub-matrice panels into memory. The 'left-looking' column-oriented variant of the factorization algorithm is implemented to reduce the disk I/O traffic. The routines are implemented using a portable I/O interface and utilize high-performance ScaLAPACK factorization routines as in-core computational kernels.**

**We present the details of the implementation for the out-of-core ScaLAPACK factorization routines, as well as performance and scalability results on a Beowulf Linux cluster. Copyright © 2000 John Wiley & Sons, Ltd.**

KEY WORDS: linear solver; out-of-core solver; LU factorization; numerical library

[*] Correspondence to: Eduardo D'Azevedo, Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831–6367, U.S.A.

Contract/grant sponsor: National Science Foundation; contract/grant number: ASC-9005933
Contract/grant sponsor: Defense Advanced Research Projects Agency; contract/grant number: DAAL03-91-C-0047
Contract/grant sponsor: Office of Scientific Computing, U.S. Department of Energy; contract/grant number: DE-AC05-00OR22725
Contract/grant sponsor: National Science Foundation Science and Technology Center Cooperative Agreement; contract/grant number: CCR-8809615
Contract/grant sponsor: U.S. Government; contract/grant number: DE-AC05-00OR22725

*Received 1998*
*Revised 26 July 2000*

## 1. INTRODUCTION

This paper describes the design and implementation of three core factorization routines—LU, QR and Cholesky—included in the out-of-core extensions of ScaLAPACK. These routines allow the factorization and solution of a dense linear system that is too large to fit entirely in physical memory.

Although current computers have unprecedented memory capacity, out-of-core solvers are still needed to tackle even larger applications. A Linux PC is commonly equipped with 512 Mbytes of memory and is capable of performing over 500 Mflops s$^{-1}$. Even on a large problem that occupies all available memory, the in-core solution of dense linear problems typically takes less than 30 minutes. On a Beowulf network of workstations (NOW) with 50 processors, it may need about two hours to solve a dense complex system of order 40 000. This suggests that the processing power of such high-performance machines is under-utilized and much larger systems can be tackled before run time becomes prohibitively large. Therefore, it is natural to develop parallel out-of-core solvers to tackle large dense linear systems. Large dense problems arise from the modeling effect of RF heating of plasmas in fusion applications [1–3] and modeling high-resolution three-dimensional wave scattering problems using the boundary element formulation [4–7]. Although a fast multipole formulation (FMM) may be an efficient alternative in some cases [8], a dense matrix formulation is still necessary in complicated geometry or when an FMM version is not available.

This development effort has the objective of producing portable software that achieves high performance on distributed memory multiprocessors, shared memory multiprocessors, and NOW. The software has been portered to run on IBM SP, Compaq Alpha cluster, SGI multiprocessors, and Beowulf Linux clusters. The implementation is based on modular software building blocks such as the PBLAS [9–11] (parallel basic linear algebra subprograms), and the BLACS [12,13] (basic linear algebra communication subprograms). Proven and highly efficient ScaLAPACK factorization routines are used for in-core computations.

Earlier out-of-core dense linear algebra efforts are reported in the literature [14–17]. A recent work [18] describes out-of-core Cholesky factorization using PLAPACK on the CRAY T3E and HP Exemplar. Our work is built upon the portable ScaLAPACK library and includes the LU, QR, and Cholesky methods. Since pivoting is required in LU factorization, the current algorithm mainly uses variable width column panels, whereas [18] is based on decomposition by square submatrices. Our work improves upon [19] in performing parallel I/O based on an in-core ScaLAPACK block-cyclic distribution. Moreover, the current implementation has more efficient handling of pivoting by storing partially pivoted factors on disk and performing an extra pass to permute the factors to final order. Another optimization technique is the use of variable width panels in the Cholesky factorization; as the factorization progresses, a wider (shorter) panel can be used in the same amount of memory. This reduces the number of passes and hence the total volume of I/O required.

One key component of an out-of-core library is an efficient and portable I/O interface. We have implemented a high-level I/O layer to encapsulate machine or architecture specific characteristics to achieve good throughput. The I/O layer eases the burden of manipulating out-of-core matrices by directly supporting the reading and writing of *unaligned* sections of ScaLAPACK block-cyclic distributed matrices.

Section 2 describes the design and implementation of the portable I/O library. The implementation of the 'left-looking' column-oriented variant of the LU, QR, and Cholesky factorization is described in Section 3. Finally, Section 4 summarizes the performance on a Beowulf Linux cluster built with common off-the-shelf components.

## 2. I/O LIBRARY

This section describes the overall design of the I/O library, including both the high-level user interface and the low-level implementation details needed to achieve good performance.

### 2.1. Low-level details

Each out-of-core matrix is associated with a device unit number (between 1 and 99), much like the familiar Fortran I/O subsystem. Each I/O operation is record-oriented, where each record is conceptually an $\mathtt{MMB} \times \mathtt{NNB}$ ScaLAPACK block-cyclic distributed matrix. Moreover, if this record/matrix is distributed with (MB, NB) as the block size on a $\mathtt{MP} \times \mathtt{NQ}$ processor grid, then $\mathrm{mod}(\mathtt{MMB}, \mathtt{MB} * \mathtt{MP}) = 0$ and $\mathrm{mod}(\mathtt{NNB}, \mathtt{NB} * \mathtt{NQ}) = 0$, i.e. MMB (and NNB) are exact multiples of $\mathtt{MB} * \mathtt{MP}$ (and $\mathtt{NB} * \mathtt{NQ}$). Data to be transferred are first copied or assembled into an internal temporary buffer (record). This arrangement reduces the number of `lseek()` system calls and encourages large contiguous block transfers, but incurs some overhead in memory-to-memory copies. All processors are involved in each record transfer. Individually, each processor writes out an (MMB/MP) by (NNB/NQ) matrix block. MMB and NNB can be adjusted to achieve good I/O performance with large contiguous block transfers or to match RAID disk stripe sizes. A drawback of this arrangement is that I/O on narrow block rows or block columns will involve only processors aligned on the same row or column of the processor grid, and thus may not obtain full bandwidth from the I/O subsystem. An optimal block size for I/O transfer may not be equally efficient for in-core computations. For example, on the Intel Paragon, MB (or NB) can be as small as 8 for good efficiency but requires at least 64 kbytes of I/O transfers to achieve good performance to the parallel file system. A *two-dimensional cyclically-shifted block layout* that achieves good load balance, even when operating on narrow block rows or block columns, was proposed in MIOS (matrix input-output subroutines) used in SOLAR [22]. However, this scheme is more complex to implement (SOLAR does not yet use this scheme). Moreover, another data redistribution is required to maintain compatibility with in-core ScaLAPACK software. A large data redistribution would incur a large message volume and a substantial performance penalty, especially in a NOW environment.

The I/O library supports both a 'shared' and 'distributed' organization of disk layout. In a 'distributed' layout, each processor opens a unique file on its local disk (e.g. '/tmp' partition on workstations) to be associated with the matrix. This is most applicable on a NOW environment or where a parallel file system is not available. On systems where a shared parallel file system is available (such as M_ASYNC mode for PFS on Intel Paragon), all processors open a common shared file. Each processor can independently perform `lseek/read/write` requests to this common file. Physically, the 'shared' layout can be the concatenation of the many 'distributed' files. Another organization is to 'interlace' contributions from individual processors into each record on the shared file. This may lead to better pre-fetch caching by the operating system, but requires an `lseek()` operation by each processor, even on reading sequential records. On the Paragon, `lseek()` is an expensive operation

Table I. Descriptor for in-core ScaLAPACK matrix.

| | Symbolic | | |
|---|---|---|---|
| DESC_() | name | Scope | Definition |
| 1 | DTYPE_A | Global | The descriptor type DTYPE_A = 1. |
| 2 | CTXT_A | Global | The BLACS context handle, indicating the BLACS process grid over which the global matrix A is distributed. The context itself is global, but the handle (the integer value) may vary. |
| 3 | M_A | Global | The number of rows in the global array A. |
| 4 | N_A | Global | The number of columns in the global array A. |
| 5 | MB_A | Global | The blocking factor used to distribute the rows of the array. |
| 6 | NB_A | Global | The blocking factor used to distribute the columns of the array. |
| 7 | RSRC_A | Global | The process row over which the first row of the array A is distributed. |
| 8 | CSRC_A | Global | The process column over which the first column of the array A is distributed. |
| 9 | LLD_A | Local | The leading dimension of the local array. LLD_A $\geq$ MAX(1, LOCp(M_A)). |

since it generates a message to the I/O nodes. Note that most implementations of NFS (Networked File System) do not correctly support multiple concurrent read/write requests to a shared file.

Unlike MIOS in SOLAR, only a synchronous I/O interface is provided for reasons of portability and simplicity of implementation. The current I/O library is written in C and uses standard POSIX I/O operations. System-dependent routines, such as NX-specific `gopen()` or `eseek()` system calls, may be required to access files over 2 Gbytes. Asynchronous I/O that overlaps computation and I/O is most effective only when processing time for I/O and computation are closely matched. Asynchronous I/O provides little benefits in cases where in-core computation or disk I/O dominates overall time. Asynchronous pre-fetch reads or delayed buffered writes also require dedicating scarce memory for I/O buffers. Having less memory available for the factorization may increase the number of passes over the matrix and increase overall I/O volume.

### 2.2. User Interface

To maintain ease of use and compatibility with existing ScaLAPACK software, a new ScaLAPACK array descriptor has been introduced. This out-of-core descriptor (`DTYPE_ = 601`) extends the existing descriptor for dense matrices (`DTYPE_ = 1`) to encapsulate and hide implementation-specific information such as the I/O device associated with an out-of-core matrix and the layout of the data on disk.

The in-core ScaLAPACK calls for performing a Cholesky factorization may consist of:

Table II. Descriptor for out-of-core matrix.

| DESC_() | Symbolic name | Scope | Definition |
|---|---|---|---|
| 1 | DTYPE_A | Global | The descriptor type DTYPE_A = 601 for an out-of-core matrix. |
| 2 | CTXT_A | Global | The BLACS context handle, indicating the MP × NQ BLACS process grid over which the global matrix A is distributed. The context itself is global, but the handle (the integer value) may vary. |
| 3 | M_A | Global | The number of rows in the global array A. |
| 4 | N_A | Global | The number of columns in the global array A. |
| 5 | MB_A | Global | The blocking factor used to distribute the rows of the MMB × NNB submatrix. |
| 6 | NB_A | Global | The blocking factor used to distribute the columns of the MMB × NNB submatrix. |
| 7 | RSRC_A | Global | The process row over which the first row of the array A is distributed. |
| 8 | CSRC_A | Global | The process column over which the first column of the array A is distributed. |
| 9 | LLD_A | Local | The conceptual leading dimension of the global array. Usually this is taken to be M_. |
| 10 | IODEV_A | Global | The I/O unit device number associated with the out-of-core matrix A. |
| 11 | SIZE_A | Local | The amount of local in-core memory available for the factorization of A. |

```
*
*  initialize descriptor for matrix A
*
            CALL DESCINIT(DESCA,M,N,MB,NB,RSRC,CSRC,ICONTXT,LDA,INFO)
*
*  perform Cholesky factorization
*
            CALL PDPOTRF(UPLO,N,A,IA,JA,DESCA,INFO)
```

where the array descriptor DESCA is an integer array of length 9 whose entries are described by Table I.

The out-of-core version is very similar:

```
*
*  initialize extended descriptor for out-of-core matrix A
*
            CALL PFDESCINIT(DESCA,M,N,MB,NB,RSRC,CSRC,ICONTXT,IODEV,
                ‘Distributed’,MMB,NNB,ASIZE, ‘/tmp/a.data’//CHAR(0),INFO)
```

```
*
*   perform out-of-core Cholesky factorization
*
               CALL PFDPOTRF(UPLO,N,A,IA,JA,DESCA,INFO)
```

where the array descriptor DESCA is an integer array of length 11 whose entries are described by Table II.

Here ASIZE is the amount of in-core buffer storage available in array ‘A’ associated with the out-of-core matrix. A ‘Distributed’ layout is prescribed and the file ‘/tmp/a.data’ is used on unit device IODEV. Each I/O record is an MMB by NNB ScaLAPACK block-cyclic distributed matrix.

The out-of-core matrices can also be manipulated by read/write calls. For example,

```
   CALL ZLAREAD(IODEV, M,N, IA,JA, B, IB,JB, DESCB, INFO)
```

reads in an M by N sub-matrix starting at position (IA,JA) into an in-core ScaLAPACK matrix B(IB:IB+M-1,JB:JB+N-1). Best performance is achieved with data transfer exactly aligned to the local processor and block boundary; otherwise redistribution by message passing may be required for unaligned non-local data transfer to matrix B.

## 3. LEFT-LOOKING ALGORITHM

The three factorization algorithms, LU, QR, and Cholesky, use a similar ‘left-looking’ organization of computation. The left-looking variant is first described as a particular choice in a block-partitioned algorithm in Section 3.1.

The actual implementation of the left-looking factorization uses two full in-core column panels (call these X, Y; see Figure 1). Panel X is NNB columns wide and panel Y occupies the remaining memory but should be at least NNB columns wide. Panel X acts as a buffer to hold and apply previously computed factors to panel Y. Once all updates are performed, panel Y is factored using an in-core ScaLAPACK algorithm. The results in panel Y are then written to disk.

The following subsections describe in more detail the implementation of LU, QR, and Cholesky factorization.

### 3.1. Partitioned factorization

The ‘left-looking’ and ‘right-looking’ variants of LU factorization can be described as particular choices in a partitioned factorization. The reader can easily generalize the following for a QR or Cholesky factorization.

Let an $m \times n$ matrix $A$ be factored into $PA = LU$ where $P$ is a permutation matrix, and $L$ and $U$ are the lower and upper triangular factors. We treat matrix $A$ as a block-partitioned matrix

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

where $A_{11}$ is a square $k \times k$ sub-matrix.

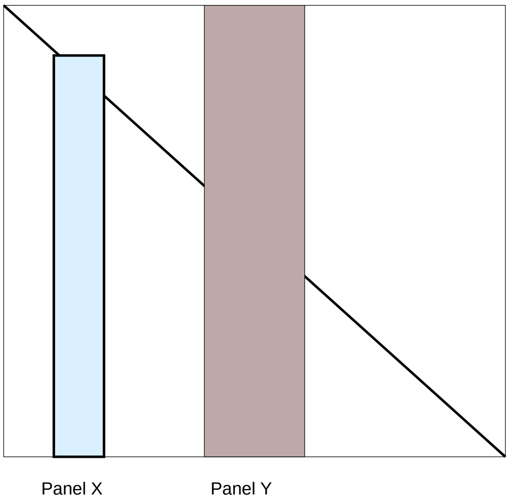

Figure 1. The algorithm requires two in-core panels.

1. The assumption is that the first $k$ columns are already factored

$$P_1 \begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix} = \begin{pmatrix} L_{11} \\ L_{21} \end{pmatrix} (U_{11}) \tag{1}$$

   where

$$A_{11} = L_{11}U_{11}, \quad A_{21} = L_{21}U_{11} \tag{2}$$

   If $k \le n_0$ is small enough, a fast non-recursive algorithm such as ScaLAPACK `PxGETRF` may be used directly to perform the factorization; otherwise, the factors may be obtained recursively by the same algorithm.

2. Apply the permutation to the unmodified sub-matrix

$$\begin{pmatrix} \tilde{A}_{12} \\ \tilde{A}_{22} \end{pmatrix} = P_1 \begin{pmatrix} A_{12} \\ A_{22} \end{pmatrix} \tag{3}$$

3. Compute $U_{12}$ by solving the triangular system

$$L_{11}U_{12} = \tilde{A}_{12} \tag{4}$$

4. Perform update to $\tilde{A}_{22}$

$$\tilde{A}_{22} \leftarrow \tilde{A}_{22} - L_{21}U_{12} \tag{5}$$

5. Recursively factor the remaining matrix

$$P_2\tilde{A}_{22} = L_{22}U_{22} \tag{6}$$

6. Final factorization is

$$P_2 P_1 \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & 0 \\ \tilde{L}_{21} & L_{22} \end{pmatrix} \begin{pmatrix} U_{11} & 0 \\ U_{12} & U_{22} \end{pmatrix}, \quad \tilde{L}_{21} = P_2 L_{21} \tag{7}$$

Note that the above is the recursively-partitioned LU factorization proposed by Toledo [20] if $k$ is chosen to be $n/2$. A right-looking variant results if $k = n_0$ is always chosen where most of the computation is the updating of

$$\tilde{A}_{22} \leftarrow \tilde{A}_{22} - L_{21} U_{12}$$

A left-looking variant results if $k = n - n_0$.

The in-core ScaLAPACK factorization routines for LU, QR, and Cholesky factorization, use a right-looking variant for good load balancing [21]. Other work has shown [19,15] that, for an out-of-core factorization, a left-looking variant generates less I/O volume compared to the right-looking variant. Toledo and Gustavson [22] shows that the recursively-partitioned algorithm ($k = n/2$) may be more efficient than the left-looking variant when a very large matrix is factored with minimal in-core storage.

### 3.2. LU factorization

The out-of-core LU factorization PFxGETRF involves the following operations.

1. If no updates are required in factorizing the first panel, all available storage is used as one panel:

   (i) LAREAD: read in part of original matrix
   (ii) PxGETRF: ScaLAPACK in-core factorization

$$\begin{pmatrix} L_{11} \\ L_{21} \end{pmatrix} (U_{11}) \leftarrow P_1 \begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix}$$

   (iii) LAWRITE: write out factors

   Otherwise, partition storage into panels X and Y.

2. We compute updates into panel Y by reading in the previous factors (NNB columns at a time) into panel X. Let panel Y hold $(A_{12}, A_{22})^t$:

   (i) LAREAD: read in part of factor into panel X
   (ii) LAPIV: physically exchange rows in panel Y to match permuted ordering in panel X

$$\begin{pmatrix} \tilde{A}_{12} \\ \tilde{A}_{22} \end{pmatrix} \leftarrow P_1 \begin{pmatrix} A_{12} \\ A_{22} \end{pmatrix}$$

   (iii) PxTRSM: triangular solve to compute upper triangular factor

$$U_{12} \leftarrow L_{11}^{-1} \tilde{A}_{12}$$

   (iv) PxGEMM: update remaining lower part of panel Y

$$\tilde{A}_{22} \leftarrow \tilde{A}_{22} - L_{21} U_{12}$$

3. Once all previous updates are performed, we apply in-core ScaLAPACK `PxGETRF` to compute LU factors in panel Y

$$L_{22}U_{22} \leftarrow P_2\tilde{A}_{22}$$

The results are then written back out to disk.

4. A final extra pass over the computed lower triangular $L$ matrix may be required to rearrange the factors in the final permutation order

$$\tilde{L}_{12} \leftarrow P_2L_{12}$$

Note that although `PFxGETRF` can accept a general rectangular matrix, a column-oriented algorithm is used. The pivot vector is held in memory and not written out to disk. During the factorization, factored panels are stored on disk with only partially or 'incompletely' pivoted row data, whereas factored panels were stored in original unpivoted form in [19] and repivoted 'on-the-fly'. The current scheme is more complex to implement but reduces the number of row exchanges required.

### 3.3. QR factorization

The out-of-core QR factorization `PFxGEQRF` involves the following operations.

1. If no updates are required in factorizing the first panel, all available memory is used as one panel:

   (i) `LAREAD`: read in part of original matrix
   (ii) `PxGEQRF`: in-core factorization

$$Q_1\begin{pmatrix}R_{11}\\0\end{pmatrix} \leftarrow \begin{pmatrix}A_{11}\\A_{21}\end{pmatrix}$$

   (iii) `LAWRITE`: write out factors.

   Otherwise, partition storage into panels X and Y.

2. We compute updates into panel Y by bringing in previous factors `NNB` columns at a time into panel X:

   (i) `LAREAD`: read in part of factor into panel X
   (ii) `PxORMQR`: apply Householder transformation to panel Y

$$\begin{pmatrix}R_{21}\\\tilde{A}_{22}\end{pmatrix} \leftarrow Q_1^t\begin{pmatrix}A_{12}\\A_{22}\end{pmatrix}$$

3. Once all previous updates are performed, we apply in-core ScaLAPACK `PxGEQRF` to compute QR factors in panel Y

$$Q_2R_{22} \leftarrow \tilde{A}_{22}$$

The results are then written back out to disk.

Note that to be compatible with the encoding of the Householder transformation in the `TAU(*)` vector as used in ScaLAPACK routines, a column-oriented algorithm is used even for rectangular matrices. The `TAU(*)` vector is held in memory and is not written out to disk.

### 3.4. Cholesky factorization

The out-of-core Cholesky factorization PxPOTRF factors a symmetric matrix into $A = LL^t$ without pivoting. The algorithm involves the following operations.

1. If no updates are required in factorizing the first panel, all available memory is used as one panel:

   (i) LAREAD: read in part of original matrix
   (ii) PxPOTRF: ScaLAPACK in-core factorization

   $$L_{11} \leftarrow A_{11}$$

   (iii) PxTRSM: modify remaining column

   $$L_{21} \leftarrow A_{21}L_{11}^{-t}$$

   (iv) LAWRITE: write out factors.

   Otherwise, partition storage into panels X and Y. We exploit symmetry by operating on only the lower triangular part of matrix $A$ in panel Y. Thus for the same amount of storage, the width of panel Y increases as the factorization proceeds.
2. We compute updates into panel Y by bringing in previous factors NNB columns at a time into panel X:

   (i) LAREAD: read in part of lower triangular factor into panel X
   (ii) PxSYRK: symmetric update to diagonal block of panel Y
   (iii) PxGEMM: update remaining columns in panel Y.

3. Once all previous updates are performed, we perform a right-looking in-core factorization of panel Y. Loop over each block column (width NB) in panel Y:

   (i) factor diagonal block on one processor using PxPOTRF
   (ii) update same block column using PxTRSM
   (iii) symmetric update of diagonal block using PxSYRK
   (iv) update remaining columns in panel Y using PxGEMM.

   Finally the computed factors are written out to disk.

Although, only the lower triangular portion of matrix A is used in the computation, the code still requires disk storage for the full matrix to be compatible with ScaLAPACK. ScaLAPACK routine PxPOTRF accepts only a square matrix distributed with square sub-blocks, MB=NB.

## 4. NUMERICAL RESULTS

Since electromagnetic scattering and fusion applications use complex*16 LU solver most heavily, we focus our attention on numerical experiments on LU factorization. The complex*16 version of

Table III. Performance of in-core ScaLAPACK computations.

| | M | $P \times Q$ | Fact (s) | Solve (s) | Mflops CPU$^{-1}$ |
|---|---|---|---|---|---|
| LU | 4500 | 7 × 8 | 74.8 | 2.4 | 58.0 |
| LU | 16 000 | 7 × 8 | 1266.4 | 9.6 | 154.0 |
| LL$'$ | 4500 | 7 × 8 | 77.8 | 1.8 | 27.9 |
| LL$'$ | 7000 | 7 × 8 | 203.3 | 3.3 | 40.2 |
| LL$'$ | 10 000 | 8 × 7 | 417.0 | 6.0 | 57.1 |
| LL$'$ | 16 000 | 7 × 8 | 824.8 | 12.5 | 118.2 |
| QR | 4500 | 7 × 8 | 95.3 | 21.4 | 91.1 |
| QR | 7000 | 7 × 8 | 255.2 | 48.8 | 128.0 |
| QR | 10 000 | 8 × 7 | 622.0 | 83.6 | 153.1 |

the prototype code[†] was tested on the `TORC II`[‡] Beowulf Linux cluster. Each node consists of a dual Pentium II at 450 Mhz with 512 Mbytes and a local 8 Gbytes IDE disk running Redhat Linux 6.2 with smp kernel. `MPIBLACS` was used with LAM/MPI[§] version 6.3 with a single 100 Mbit s$^{-1}$ ethernet connection per node. Two MPI tasks per node were spawned to fully utilize both processors. A single CPU can achieve about 320 Mflops s$^{-1}$ in ZGEMM operations with optimized BLAS libraries produced by ATLAS[¶]. The experiments were performed with `MB=NB=50`, `NRHS=10` for solution with 10 vectors, and 10 000 000 words (160 Mbytes) per task was allocated to the out-of-core software. The out-of-core arrays were stored on the local disk using the `'DISTRIBUTED'` option.

As in [18], we report performance in Mflops s$^{-1}$ CPU$^{-1}$. Table III shows the performance of in-core ScaLAPACK solvers. The results show that performance increases with problem size to about 154 Mflops s$^{-1}$ CPU$^{-1}$. Note that the lower performance for Cholesky (`LL'`) is due to the high proportion of work performed in triangular solves (`PxTRSM`).

Table IV shows the performance of out-of-core ScaLAPACK on various problem sizes and processor grid configurations. Note that if sufficient in-core storage is available, the library will by-pass the panel algorithm and revert to the in-core ScaLAPACK routines. The 'fact' time is the total elapsed time (including I/O) to perform out-of-core factorization; the 'solve' time is the total elapsed time (including I/O) for solution with 10 vectors. The 'read' and 'write' times are the total accumulated elapsed time spent in I/O routines on processor 0. As observed in [18], the performance for the out-of-core solver (190 Mflops s$^{-1}$ CPU$^{-1}$) is higher than that for the in-core solver since most of the computation is performed in large blocks. On the largest problem ($N = 80\,000$) that took 35.6 h to perform the LU

[†] Available from `http://www.netlib.org/scalapack/prototype`.

[‡] `http://www.epm.ornl.gov/torc/`. Special thanks to Stephen Scott for arranging dedicated use of this cluster.

[§] `http://www.mpi.nd.edu/lam`.

[¶] `http://www.netlib.org/atlas/index.html`.

Table IV. Results of out-of-core computations.

| | $M$ | $P \times Q$ | Fact (s) | Solve (s) | Read (s) | Write (s) | Mflop CPU$^{-1}$ |
|---|---|---|---|---|---|---|---|
| LU | 16 000 | 4 × 4 | 3670.0 | 180.1 | 276.3 | 141.4 | 186.0 |
| QR | 16 000 | 4 × 4 | 8139.5 | 549.0 | 234.1 | 115.7 | 167.7 |
| LL′ | 16 000 | 4 × 4 | 2815.8 | 222.7 | 259.9 | 103.6 | 121.2 |
| LU | 32 000 | 4 × 4 | 28 850.2 | 508.2 | 1303.7 | 614.8 | 189.3 |
| LU | 16 000 | 4 × 8 | 1866.9 | 123.4 | 114.2 | 48.4 | 182.8 |
| QR | 16 000 | 4 × 8 | 3520.3 | 506.1 | 149.6 | 57.2 | 193.9 |
| LL′ | 16 000 | 4 × 8 | 1878.5 | 100.1 | 126.8 | 51.8 | 90.9 |
| LU | 32 000 | 4 × 8 | 15 730.4 | 312.8 | 634.1 | 316.6 | 173.6 |
| LU | 40 000 | 4 × 8 | 32 420.1 | 442.5 | 1100.3 | 494.8 | 164.5 |
| LU | 20 000 | 8 × 7 | 2301.4 | 81.5 | 23.4 | 54.2 | 165.5 |
| QR | 20 000 | 8 × 7 | 3917.0 | 405.6 | 41.0 | 44.3 | 194.5 |
| LL′ | 20 000 | 8 × 7 | 2072.3 | 77.3 | 42.1 | 46.1 | 91.9 |
| LU | 32 000 | 8 × 7 | 8316.7 | 288.6 | 310.2 | 160.0 | 187.6 |
| LU | 45 000 | 8 × 7 | 22 508.2 | 481.5 | 729.5 | 418.0 | 192.8 |
| LL′ | 45 000 | 8 × 7 | 17 946.0 | 332.9 | 384.3 | 191.2 | 120.9 |
| LU | 80 000 | 8 × 7 | 128 154.9 | 898.7 | 2352.1 | 1275.1 | 190.2 |

factorization on 28 nodes (10.6 Gflops s$^{-1}$), each MPI task created a 1.8 Gbytes[‖] local file. The time for I/O (about 3627 s)[**] was a small fraction of overall time. This suggests the out-of-core computation on this machine is compute bound and overlapping computation with asynchronous I/O may not produce significant benefits.

## 5. CONCLUSIONS

The out-of-core ScaLAPACK extension provides an easy to use interface similar to the in-core ScaLAPACK library. The software is portable and achieves high performance even on a Beowulf Linux PC cluster using off-the-shelf components. The software allows very large problems of several times total available memory to be solved with high performance.

**ACKNOWLEDGEMENTS**

This work was supported in part by the National Science Foundation Grant No. ASC-9005933; by the Defense Advanced Research Projects Agency under contract DAAL03-91-C-0047, administered by the Army Research

[‖] 2 Gbytes is the maximum file size under the Linux ext2 file system.

[**] Effective I/O throughput about 2.8 Mbytes s$^{-1}$ CPU$^{-1}$.

Office; by the Office of Scientific Computing, U.S. Department of Energy, under Contract DE-AC05-00OR22725; and by the National Science Foundation Science and Technology Center Cooperative Agreement No. CCR-8809615, and Center for Computational Sciences at Oak Ridge National Laboratory for the use of the computing facilities. The submitted manuscript has been authored by a contractor of the U.S. Government under contract DE-AC05-00OR22725. Accordingly, the U.S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes.

## REFERENCES

1. Berry LA, Jaeger EF, Batchelor DB. Wave-induced momentum transport and flow drive in tokamak plasmas. *Physics Review Letters* 1999; **82**:1871.
2. Jaeger EF, Berry LA, Batchelor DB. Second-order radio frequency kinetic theory with applications to flow drive and heating in tokamak plasmas. *Physics of Plasmas* 2000; **7**:641.
3. Jaeger EF, Berry LA, Batchelor DB. Full-wave calculation of sheared poloidal flow driven by high harmonic ion Bernstein waves in tokamak plasmas. *Physics of Plasmas* 2000; **7**(8):3319–3329.
4. Cwik T, van de Geijn R, Patterson J. The application of parallel computation to integral equation models of electromagnetic scattering. *Journal of the Optical Society of America A* 1994; **11**(4):1538.
5. Demkowicz L, Karafiat A, Oden JT. Solution of elastic scattering problems in linear acoustics using h-p boundary element method. *Computer Methods in Applied Mechanics and Engineering* 1992; (101):251.
6. Geng P, Oden JT, van de Geijn R. Massively parallel computation for acoustical scattering problems using boundary element methods. *Journal of Sound and Vibration* 1996; **191**(1):145.
7. Semeraro BD, Gray LJ. PVM implementation of the symmetric-Galerkin method. *Engineering Analysis with Boundary Elements* 1997; **19**(1):67.
8. Fu Y, Klimkowski KJ, Rodin GJ, Berger E, Browne JC, Singer JK, van de Geijn RA, Vemaganti KS. A fast solution method for three-dimensional many-particle problems of linear elasticity. *International Journal of Numerical Methods in Engineering* 1998; **42**:1215.
9. Choi J, Dongarra J, Ostrouchov S, Petitet A, Walker D, Whaley RC. A proposal for a set of parallel basic linear algebra subprograms. *Technical Report CS-95-292*, University of Tennessee, Knoxville, TN, May 1995 (also available as LAPACK Working Note #100).
10. Choi J, Dongarra J, Walker D, PB-BLAS: A set of parallel block basic linear algebra subroutines. *Concurrency: Practice and Experience* 1996; **8**:517–535.
11. Petitet A. Algorithmic redistribution methods for block cyclic decompositions. *PhD Thesis*, University of Tennessee, Knoxville, Tennessee, 1996 (also available as LAPACK Working Note # 128 and #133).
12. Dongarra J, van de Geijn RA. Two dimensional basic linear algebra communication subprograms. *Technical Report CS-91-138*, University of Tennessee, Knoxville, Tennessee, 1991 (also available as LAPACK Working Note #37).
13. Dongarra J, van de Geijn RA. Whaley RC. Two dimensional basic linear algebra communication subprograms. *Environments and Tools for Parallel Scientific Computing*, Vol 6. Elsevier Science Publishers B.V., 1993; 31–40.
14. Brunet J-P, Pederson P, Johnsson SL. Load-balanced LU and QR factor and solve routines for scalable processors with scalable I/O. *Proceedings of the 17th IMACS World Congress*, 1994.
15. Klimkowski K, van de Geijn RA. Anatomy of a parallel out-of-core dense linear solver. *Proceedings of the International Conference on Parallel Processing*, 1995.
16. Scott DS. Out of core dense solvers on Intel parallel supercomputers. *Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation*, 1992; 484.
17. Scott DS. Parallel I/O and solving out-of-core systems of linear equations. *Proceedings of the 1993 DAGS/PC Symposium*, Darmouth Institute for Advanced Graduate Studies, 1993; 123.
18. Reiley WC, van de Geijn RA. POOCLAPACK: Parallel out-of-core linear algebra package. *Technical Report 99-33*, Department of Computer Science, The University of Texas, Austin, Texas, 1999 (also available as PLAPACK Working Note #10).
19. Dongarra J, Hammarling S, Walker D. Key concepts for parallel out-of-core LU factorization. *Computers and Mathematics with Applications* 1998; **35**(7):13–31.
20. Toledo S. Locality of reference in LU decomposition with partial pivoting. *Technical Report RC 20344(1/19/96)*, IBM Research Division, T. J. Watson Research Center, Yorktown Heights, New York, 1996.
21. Choi J, Dongarra JJ, Ostrouchov LS, Petitet AP, Walker DW, Whaley RC. The design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines. *Technical Report ORNL/TM-12470*, Oak Ridge National Laboratory, 1994.
22. Toledo S, Gustavson F. The design and implementation of SOLAR, a portable library for scalable out-of-core linear algebra computations. *IOPADS Fourth Annual Workshop on Parallel and Distributed I/O*. ACM Press, 1996; 28–40.