THE DESIGN AND IMPLEMENTATION OF THE PARALLEL OUT-OF-CORE SCALAPACK LU, QR AND CHOLESKY FACTORIZATION ROUTINES *

ED F. D'AZEVEDO† AND JACK J. DONGARRA‡

Abstract. This paper describes the design and implementation of three core LU, QR and Cholesky factorization routines included in the out-of-core extension of ScaLAPACK. These routines allow the factorization and solution of a very large dense system that is too large to fit entirely in physical memory. An image of the full matrix is maintained on disk and the factorization routines transfer sub-matrices to be operated in memory. A '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 uses high performance ScaLAPACK factorization routines as in-core computational kernels.

We present the details of the implementation of the out-of-core ScaLAPACK factorization routines as well as performance and scalability results on the Intel Paragon.

Key words. Linear solver, out-of-core solver, LU factorization, numerical library

1. Introduction. This paper describes the design and implementation of three core LU, QR and Cholesky factorization routines included in the out-of-core extensions of ScaLAPACK. These routines allow the factorization and solution of a very large dense 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 ever larger applications. A modern workstation is commonly equipped with 64 to 128Mbytes of memory and capable of performing over 100 Mflops/sec. Even on a large problem that occupies all available memory, the in-core solution of dense linear problems typically takes less than an hour. On a network of workstations (NOW) with 100 processors, each with 64Mbytes, it may require about 30 minutes to factor and solve at 64-bit precision a dense linear system of order 30,000. This suggests that the processing power of such high performance machines are 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. Such dense problems arise from high resolution three-dimensional electromagnetic scattering problems or in modeling fluid flow past complex objects.

The development effort has the objective of producing portable software that achieves high performance on distributed memory multiprocessors, shared memory multiprocessors and NOW. The implementation is based on modular software building blocks such as PBLAS (Parallel Basic Linear Algebra Subroutines), and BLACS (Basic Linear Algebra Communication Subroutines). Proven and highly efficient ScaLAPACK factorization routines are used for in-core computations.

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.

* Research supported by DARPA / NFS / DOE and CCS-ORNL for the use of the computing facilities.

†Mathematical Sciences Section, Oak Ridge National Laboratory, e6d@ornl.gov

‡Department of Computer Science, University of Tennessee, Knoxville, dongarra@cs.utk.edu

Section 2 describes the design and implementation of the portable I/O Library. The implementation of the 'left-looking' column-oriented variant of LU, QR and Cholesky factorization is described in §3. Finally, §4 summarizes the performance on the Intel Paragon.

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 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 a $\mathtt{MMB}\times\mathtt{NNB}$ ScaLAPACK block-cyclic distributed matrix. Moreover if this record/matrix is distributed with (MB,NB) as the block size on a $\mathtt{MP}\times\mathtt{NB}$ processor grid, then $\mathrm{mod}(\mathtt{MMB},\mathtt{MB}*\mathtt{MP})=0$ and $\mathrm{mod}(\mathtt{NNB},\mathtt{NB}*\mathtt{NP})=0$, i.e. MMB (and NNB) are exact multiples of $\mathtt{MB}*\mathtt{MP}$ (and $\mathtt{NB}*\mathtt{NP}$). Data to be transfered is 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 a (MMB/MP) by (NNB/NP) 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 size. 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 on the processor grid and thus may not obtain full bandwidth from the I/O subsystem.

The MIOS (Matrix Input-Output Subroutines) used in SOLAR (Scalable Out-of-Core Linear Algebra Routines) [5] is less flexible in requiring that (MMB, NNB) equals (MB, NB). An optimal block size for I/O transfer may not be equally efficient for in-core computations. On the Intel Paragon, MB (or NB) can be as small as 8 for good efficiency but requires at least 64Kbytes I/O transfers to achieve good performance to the parallel file system. A *2-dimensional cyclically-shifted block layout* that achieves good load balance even when operating on narrow block rows or block columns was proposed in MIOS. 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 since it generates a message to the I/O nodes. Note that most implementation 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. A fully portable (although possibly not the most efficient) implementation of the I/O layer using Fortran record-oriented I/O is also possible[1]. 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 2Gbytes. 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 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, the ScaLAPACK matrix descriptor field is extended to encapsulate and hide implementation-specific information such as the I/O device associated with an out-of-core matrix and the layout of data on disk.

The in-core ScaLAPACK calls for performing an LU factorization may consist of:

```
!
!  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)
```

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,
   'SHARED',MMB,NNB,ASIZE, '/pfs/a.data'//CHAR(0),INFO)
!
!  perform out-of-core Cholesky factorization
!
CALL PFDPOTRF(UPLO,N,A,IA,JA,DESCA,INFO)
```

Here `ASIZE` is amount of in-core buffer storage available in array 'A' associated with the out-of-core matrix. A `'Shared'` layout is prescribed and the file `'/pfs/A.data'` is used on unit device `IODEV`. Each I/O record is a `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 a `M` by `N` sub-matrix starting at `(IA,JA)` position into an in-core ScaLAPACK

[1] We are not aware of any implementation of fully portable asynchronous I/O short of using threads. However, a portable thread library may not be available and greatly complicates the code.

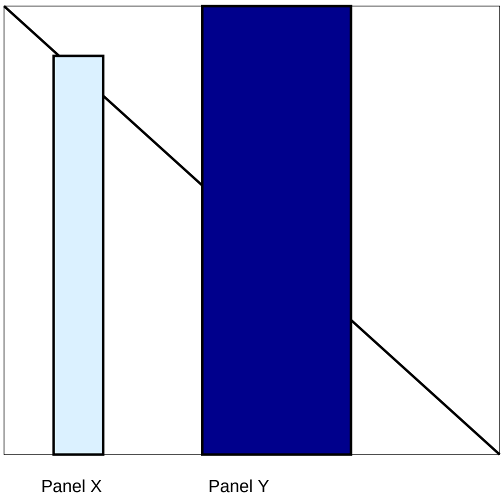

FIG. 3.1. *Algorithm requires 2 in-core panels.*

matrix `B(IB:IB+M-1,JB:JB+N-1)`. Best performance is achieved with data transfer aligned to local processor and block boundary; otherwise message passing is performed for unaligned non-local data transfer to matrix `B`.

3. Left-looking Algorithm. The three factorization algorithms, LU, QR, and Cholesky, all 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 §3.1.

The actual implementation of the left-looking factorization uses two full column in-core panels (call these X, Y; see Figure 3.1). 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 back out 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 partition 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.

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

$$P_1 \begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix} = \begin{pmatrix} L_{11} \\ L_{21} \end{pmatrix} (U_{11}) \quad , \tag{3.1}$$

where

$$A_{11} = L_{11} U_{11}, \quad A_{21} = L_{21} U_{11} \ . \tag{3.2}$$

If $k \leq n_0$ 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

$$\begin{pmatrix} \tilde{A}_{12} \\ \tilde{A}_{22} \end{pmatrix} = P_1 \begin{pmatrix} A_{12} \\ A_{22} \end{pmatrix} . \tag{3.3}$$

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

$$L_{11}U_{12} = \tilde{A}_{12} \tag{3.4}$$

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

$$\tilde{A}_{22} \leftarrow \tilde{A}_{22} - L_{21}U_{12} \tag{3.5}$$

5. Recursively factor the remaining matrix

$$P_2\tilde{A}_{22} = L_{22}U_{22} \tag{3.6}$$

6. Final factorization is

$$P_2P_1 \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & 0 \\ \tilde{L}_{21} & L_{22} \end{pmatrix} \begin{pmatrix} U_{11} & 0 \\ U_{12} & U_{22} \end{pmatrix}, \quad \tilde{L}_{21} = P_2 L_{21} . \tag{3.7}$$

Note that the above is the recursively-partitioned LU factorization proposed by Toledo [4] 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, all use a right-looking variant for good load balancing [1]. Other work has shown [2, 3] that for out-of-core factorization, a left-looking variant generates less I/O volume compared to the right-looking variant. Toledo [5] shows that the recursively-partitioned algorithm ($k = n/2$) may be more efficient than the left-looking variant for very large matrices and solved with minimal in-core storage.

3.2. LU Factorization. The out-of-core LU factorization `PxGETRF` 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 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

$$\tilde{L}_{12} \leftarrow P_2 L_{12} \ .$$

Note that although `GETRF` 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 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 [2] 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 `GEQRF` 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

$$Q_2 R_{22} \leftarrow \tilde{A}_{22}$$

The results are then written back out to disk.

Note that to be compatible with the encoding of Householder transformation in the `TAU(*)` vector as used 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. The prototype code is still under active development and testing[2]. The double precision version was tested on the Intel Paragon systems at the Center for Computational Sciences, Oak Ridge National Laboratory. The xps35 has 512 GP nodes organized in a 16 row by 32 column rectangular mesh. Each GP

[2]The prototype code is available from `http://www.netlib.org/scalapack/prototype`

TABLE 4.1
Performance of out-of-core LU factorization on 64 processors using MB=NB=50.

size of matrix	lwork (doubles)	update (sec)	fact (sec)	reorder (sec)	total (sec)	in-core (processors)
5000	130000	38	28	18	151	59 (64)
8000	250000	126	60	49	389	180 (64)
10000	375000	231	95	74	640	130 (256)
16000	1000000	858	301	192	1946	388 (256)
20000	1000000	1782	377	290	3502	681 (256)

TABLE 4.2
Performance of out-of-core QR factorization on 64 processors using MB=NB=50.

size of matrix	lwork (doubles)	update (sec)	fact (sec)	total (sec)	in-core (processors)
5000	130000	78	41	176	92 (64)
8000	260000	271	98	516	310 (64)
10000	410000	496	161	900	200 (256)
16000	1000000	1816	536	2893	647 (256)
20000	1000000	3805	680	5466	1176 (256)

node has 32MBytes of memory. The xps150 has 1024 MP nodes organized in a 16 row by 64 column rectangular mesh. Each MP node has at least 64MBytes of memory. The MP node has 2 compute cpu's to support multi-threaded code, but to make results comparable to xps35, only one cpu was utilized in the test. The runs were performed in a multiuser (non-dedicated) environment. Runs on 64 (256) processor were performed on xps35 (xps150) using a 8×8 (16×16) *logical* processor grid. The xps150 was used to ensure in-core solves of the large matrices are resident in memory without page faults to disk.

Initial experiments suggest that I/O performance may vary by a wide margin and depends on the I/O and paging requests in other applications. The double precision version was tested with block size of MB = NB = 50, MMB = 800 and NNB = 400. A shared file was used on '/pfs' parallel file system (16-way interleaved RAID system with 64Kbyte stripes). The shared file was opened with NX-specific `M_ASYNC` mode in the `gopen()` system call.

Table 4.1 shows the run time (in seconds) for out-of-core LU factorization on the Intel Paragon. Field *lwork* is amount of temporary storage (number of double precision numbers) available to the out-of-core routine for panels X and Y. Field *update* is the computation time (excluding I/O) for `PxTRSM` and `PxGEMM` updates from panel X to panel Y. Field *fact* is the total computation time (excluding I/O) required to factor panel Y. Field *reorder* is the total time for I/O and `PxLAPIV` to reorder the lower triangular factors into the final pivoted order. Field *in-core* shows the computation time (and number of processors used) for an all in-core factorization using ScaLAPACK `PDGETRF` routine.

We are considering streamlining the out-of-core PFxGETRF LU factorization code (and PFxGETRS right-hand solver) to leave the lower factors in partially pivoted form and avoid the extra pass required to reorder the lower triangular matrix into final pivoted order. Note that without this extra reordering cost, the out-of-core solver incurs approximately a 18% overhead over in-core solvers $((3502 - 290)/(681 * 4) \approx 1.18)$.

Table 4.2 shows the run time (in seconds) for out-of-core QR factorization on the Intel Paragon. Field *lwork* is amount of temporary storage (number of double precision

TABLE 4.3
Performance of out-of-core Cholesky factorization on 64 processors using MB=NB=50.

size of matrix	lwork (doubles)	update (sec)	fact (sec)	total (sec)	in-core (processors)
5000	130000	20	18	77	39 (64)
8000	260000	56	45	196	90 (64)
10000	410000	93	78	311	60 (256)
16000	1000000	339	264	937	191 (256)
20000	1000000	776	354	1655	340 (256)

numbers) available to the out-of-core routine for panels X and Y. Field *update* is the computation time (excluding I/O) for Householder updates using `PxORMQR` from panel X to panel Y. Field *fact* is the total computation time (excluding I/O) required to factor panel Y using `PxGEQRF`. Field *in-core* shows the computation time (and number of processors used) for an all in-core factorization using ScaLAPACK `PDGEQRF` routine. For large problems, the out-of-core version incurs an overhead of around 16% over the in-core solver ($(5466/4)/1176 \approx 1.16$).

Table 4.3 shows the run time (in seconds) for out-of-core Cholesky factorization on the Intel Paragon. Field *lwork* is amount of temporary storage (number of double precision numbers) available to the out-of-core routine for panels X and Y. Field *update* is the computation time (excluding I/O) for `PxSYRK` and `PxGEMM` updates from panel X to panel Y. Field *fact* is the total computation time (excluding I/O) required to factor panel Y. Field *in-core* shows the computation time (and number of processors used) for an all in-core factorization using ScaLAPACK `PDPOTRF` routine. For large problems, the out-of-core version incurs about a 22% overhead over the in-core version ($(1655/4)/340 \approx 1.22$).

Effectiveness of the out-of-core solvers depends in part on the amount of available core memory and on the performance of the I/O system. The results on the xps35 suggest that the out-of-core solvers are most effective on very large problems greater than available core memory and incurs about a 20% penalty over the in-core solvers.

REFERENCES

[1] J. Choi, J. J. Dongarra, L. S. Ostrouchov, A. P. Petitet, D. W. Walker, and R. C. Whaley, *The design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines*, Tech. Report ORNL/TM-12470, Oak Ridge National Laboratory, 1994.

[2] J. Dongarra, S. Hammarling, and D. Walker, *Key concepts for parallel out-of-core LU factorization*, SIAM Press, 1996. (also LAPACK Working Note # 110).

[3] K. Klimkowski and R. A. van de Geijn, *Anatomy of a parallel out-of-core dense linear solver*, in Proceedings of the International Conference on Parallel Processing, 1995.

[4] S. Toledo, *Locality of reference in lu decomposition with partial pivoting*, Tech. Report RC 20344(1/19/96), IBM Research Division, T. J. Watson Research Center, Yorktown Heights, New York, 1996.

[5] S. Toledo and F. Gustavson, *The design and implementation of SOLAR, a portable library for scalable out-of-core linear algebra computations*, in IOPADS Fourth Annual Workshop on Parallel and Distributed I/O, ACM Press, 1996, pp. 28–40.