Symmetric Pivoting in ScaLAPACK

Craig Lucas
University of Manchester

May, 2006

ABSTRACT: Recently codes have been developed for computing the Cholesky factorization with complete pivoting of a symmetric positive semidefinite matrix for the serial LAPACK library. In the parallel ScaLAPACK library there are only routines for the unpivoted factorization in the positive definite case and no algorithms use complete pivoting. We aim to assess the feasibility of complete pivoting in ScaLAPACK by implementing a parallel pivoted Cholesky routine. We discuss the steps needed to parallelize the existing serial code, and discuss the specific constraints of the data distribution and communication for ScaLAPACK. We present some experiments, comparing our code and the existing ScaLAPACK code, conducted on both a Cray XD1 and a Cray XT3. We show that on fewer processors our new code scales well and the pivoting overhead is small. However, the pivoting overhead increases with the number of processors, but decreases with problem size.

KEYWORDS: dense linear algebra, Cholesky factorization, complete pivoting, ScaLAPACK, parallel distributed algorithms

1 Introduction

The Cholesky factorization of a symmetric positive definite matrix $A \in \mathbb{R}^{n\times n}$ has the form

$$A = LL^T,$$

where $L \in \mathbb{R}^{n\times n}$ is a lower triangular matrix with positive diagonal elements. If A is positive *semi*definite, of rank r, there exists a Cholesky factorization with *complete pivoting* ([8, Thm. 10.9], for example). That is, there exists a permutation matrix $P \in \mathbb{R}^{n\times n}$ such that

$$P^T AP = LL^T,$$

where L is unique in the form

$$L = \begin{bmatrix} L_{11} & 0 \\ L_{12} & 0 \end{bmatrix},$$

with $L_{11} \in \mathbb{R}^{r\times r}$ lower triangular with positive diagonal elements. L is such that

$$\ell_{11} \geq \ell_{22} \geq \cdots \geq \ell_{rr}.$$

A common occurrence of positive semidefinite matrices is in statistics, namely covariance matrices. The (i, j)th element of a covariance matrix, S, holds the sample covariance of the ith and jth random variable.

The factorization is also used in some algorithms solving the linear least squares problem

$$\min_x \|Ax - b\|_2, \quad A \text{ semidefinite},$$

and as a test for whether a matrix is numerically positive semidefinite or not.

In the ScaLAPACK [2] (Scalable Linear Algebra PACKage) the routines `PxPOTRF` perform the Cholesky factorization of a (dense) positive definite matrix without any pivoting. We seek to add the functionality of dealing with semidefinite matrices by implementing a parallel algorithm with complete pivoting. Our algorithm is based on the blocked serial algorithm of [9], written for LAPACK [1]. There are currently no algorithms in ScaLAPACK that use complete pivoting.

In this paper we describe the parallelization of the algorithm in [9]. We also report on some numerical experiments with a code that is still under development. We look at speed, scalability and the overall feasibility of complete pivoting with the constraints of the ScaLAPACK data distribution and existing auxiliary routines.

2 ScaLAPACK

Data in ScaLAPACK is distributed in a *block cyclic* manner according to a BLACS [5] (Basic Linear Algebra Communication Subroutines) process grid, a conceptual rectangular arrangement of processes. The BLACS provide routines for frequently occurring operations in linear algebra. They are portable and vendors supply machine specific versions built on appropriate communication routines such as MPI.

Block cyclic distribution ensures good load balancing and aids scalability, and there is analysis to support this. The data is distributed in a round robin way for blocks of size $MB \times NB$.

This is best illustrated by an example. Figure 1 shows a 2-by-4 BLACS process grid. The numbers around the edge indicate the coordinates of each process, with the process number shown on the different coloured squares for each processes. That is, process number 2 is at coordinate $(0, 2)$ in the process grid.

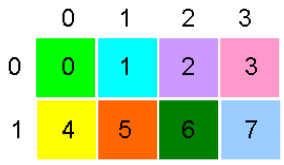

Figure 1: A 2-by-4 BLACS process grid.

If we distribute a two dimensional array according to the process grid in Figure 1, in a block cyclic manner for a block size of $MB \times NB$, then we have the situation shown on the left hand side of Figure 2. The data stored by a process is coloured according to the processes in Figure 1. The right hand side of Figure 2 shows the local storage of the parts of the array on the process with coordinates $(0, 0)$. The array is stored in column major order. The column highlighted is stored contiguously in memory, in the direction of the arrow.

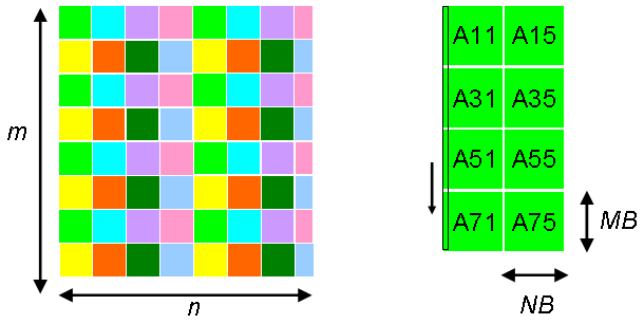

Figure 2: A two dimensional array distributed according to a 2-by-4 process BLACS grid, with block size $MB \times NB$.

The computational routines ScaLAPACK is built on top of LAPACK and the PBLAS [3] (Parallel Basic Linear Algebra Subprograms).

3 A Blocked Serial Algorithm

Our algorithm is a *blocked* algorithm. That is an algorithm that treats the matrix as consisting of not individual elements but blocks of elements. In the following algorithm we use a *block update*, which involves matrix multiplication type operations. Here, after a number of steps, we update the trailing matrix and then restart the factorization from this smaller updated matrix. These matrix-matrix operations are more efficient in terms of data reuse and use the memory hierarchy of a machine more effectively. This leads to more floating point operations per second than the matrix-vector type operations in the rest of the algorithm. The algorithm we present in this section has been shown to compute up to 8 times faster than its unblocked counterpart [9]. For a detailed discussion on blocked algorithms see [1].

The basic steps of our algorithm are based on a GAXPY (Generalized Ax Plus y) algorithm and are:

Set L = lower triangular part of A
for $j = 1{:}\,n$
  $\ell_{jj} = \ell_{jj} - L_{j1}L_{j1}^T$
  $\ell_{jj} = \sqrt{\ell_{jj}}$
  % Update jth column
  if $1 < j < n$ $L_{3j} = L_{3j} - L_{31}L_{j1}^T$
  if $j < n$ $L_{32} = L_{3j}/\ell_{jj}$
end

We use the following block representation to give us the block update. We can write for the semidefinite matrix $A^{(k-1)} \in \mathbb{R}^{n\times n}$ and $n_b \in \mathbb{R}$ [7]

$$\begin{aligned} A^{(k-1)} &= \begin{bmatrix} A_{11}^{(k-1)} & A_{12}^{(k-1)} \\ A_{12}^{T(k-1)} & A_{22}^{(k-1)} \end{bmatrix} \\ &= \begin{bmatrix} L_{11} & 0 \\ L_{21} & I_{n-n_b} \end{bmatrix} \begin{bmatrix} I_{n_b} & 0 \\ 0 & A^{(k)} \end{bmatrix} \begin{bmatrix} L_{11} & 0 \\ L_{21} & I_{n-n_b} \end{bmatrix}^T, \end{aligned}$$

where $L_{11} \in \mathbb{R}^{n_b\times n_b}$ and $L_{21} \in \mathbb{R}^{(n-n_b)\times n_b}$ form the first n_b columns of the Cholesky factor L of $A^{(k-1)}$. Now to complete our factorization of $A^{(k-1)}$ we need to factor the reduced matrix

$$A^{(k)} = A_{22}^{(k-1)} - L_{21}L_{21}^T, \tag{3.1}$$

which we can explicitly form, taking advantage of symmetry.

Thus we perform n_b steps of the GAXPY algorithm above, update the trailing matrix, than restart the GAXPY process on this smaller matrix.

We can add pivoting by looking for the largest possible value of ℓ_{jj} at each step. We call this the pivot. This is implemented in the following algorithm:

Algorithm 3.1 *This algorithm computes the pivoted Cholesky factorization with complete pivoting $P^TAP = LL^T$ of a symmetric positive semidefinite matrix $A \in \mathbb{R}^{n\times n}$, overwriting A with L, using a block update and block size n_b. The nonzero elements of the permutation matrix P are given by $P(piv(k), k) = 1$, $k = 1\colon n$.*

Set L = lower triangular part of A
$\epsilon = nu$
$piv = 1\colon n$
% Tolerance in stopping criterion
$tol = n * u * \max(\mathrm{diag}(A))$
for $k = 1\colon n_b\colon n$
  % Allow for last incomplete block
  $jb = \min(n_b, n - k + 1)$
  % Store accumulated dot products
  $dots(k\colon n) = 0$
  for $j = k\colon k + jb - 1$
    if $j > k$
      $dots(i) = dots(i) + L(i, j-1)^2, \quad i = j\colon n$
    end
    $q = \min\Big\{p : L(p,p) - dots(p) = \max_{j\le i\le n}\{L(i,i) - dots(i)\}\Big\}$
    if $L(q,q) \le tol$
      % computed rank of A is $j-1$
      return
    end
    swap $L(j,:)$ and $L(q,:)$
    swap $L(:,j)$ and $L(:,q)$
    swap $dots(j)$ and $dots(q)$
    swap $piv(j)$ and $piv(q)$
    $L(j,j) = L(j,j) - dots(j)$
    $L(j,j) = \sqrt{L(j,j)}$
    % Update jth column
    if $1 < j < n$
      $L(j+1\colon n, j) = L(j+1\colon n, j) - L(j+1\colon n, 1\colon j-1)L(j, 1\colon j-1)^T$
    end
    if $j < n$
      $L(j+1\colon n, j) = L(j+1\colon n, j)/L(j,j)$
    end
  end
  if $k + jb < n$
  % perform block update
    $L(j+1\colon n, j+1\colon n) = L(j+1\colon n, j+1\colon n) - L(j+1\colon n, 1\colon j)L(j+1\colon n, 1\colon j)^T$
  end
end

This algorithm requires $n^3/3$ flops. Note for computational efficiency we can store the inner products when calculating the pivot and update them on each iteration. The pivoting overhead is $3(r+1)n - 3/2(r+1)^2$ flops and $(r+1)n - (r+1)^2/2$ comparisons, where $r = \mathrm{rank}(A)$.

The computed rank of A is determined by stopping when a pivot is less than or equal to *tol*, which is a tolerance. For a discussion of this and a full derivation of this algorithm and the LAPACK style code see [9].

4 A Block Cyclic Parallel Algorithm

In parallelizing Algorithm 3.1 we do the following.

Firstly, as Algorithm 3.1 is blocked it fits into block cyclic data distribution. We use a square block size, $MB = NB$. The GAXPY part of the algorithm on the *current block column* is computed by a *process column* in the BLACS grid, the one which owns that part of the distributed matrix. All processes take part in the block update.

Secondly, there is no way in ScaLAPACK to extract the diagonal elements of the distributed matrix. At the start and after each block update we send the diagonal elements, diagonal block by diagonal block, to the processors in the current process column in the sender's process row to some temporary workspace. After this the current process column has a distributed copy of the diagonal elements in the trailing matrix stored as a vector. We keep this and update it at each stage of factorizing the current block column. We use the BLACS routine `DGEBS2D` which sends a rectangular array. We call the routine as if we are sending the first row in a block, but we give the overall number of rows in the local array as one more, this picks out the diagonal elements.

At each step of the current column we need to compute the pivot. Each processor computes its contribution, without the need for further communication, and stores this in a workspace array. We can then use the PBLAS routine `PDAMAX` on this workspace array to compute the maximum value. The routine returns the pivot to each processor in the current column, each process then broadcasts the pivot along their process row. Clearly there is a lot more communication between processes required in a pivoted algorithm compared to one without pivoting.

With the pivot information received, we then need to swap the current and pivot rows and columns. We facilitate the swaps with the PBLAS routine `PDSWAP`.

The vector *piv* which stores the information needed to compute the permutation matrix is distributed over *every* process column and rows are swapped when a pivot is determined.

Finally, computing the current column of L is done by calling the equivalent PBLAS routines to the BLAS [6] [4] routines in our LAPACK routine. Likewise the block update.

Implementing these considerations we have the following algorithm:

Algorithm 4.1 *This algorithm computes the pivoted Cholesky factorization with complete pivoting $P^TAP = LL^T$ of a symmetric positive semidefinite matrix $A \in \mathbb{R}^{n\times n}$, overwriting A with L. The algorithm requires data to be distributed over a BLACS process grid in a block cyclic manner with a block size of $n_b \times n_b$. The nonzero elements of the permutation matrix P are given by $P(piv(k), k) = 1$, $k = 1\colon n$.*

Set L = lower triangular part of A
$\epsilon = nu$
piv is distributed over each process column
$piv = 1\colon n$
% Tolerance in stopping criterion
$tol = n * u * \max(\mathrm{diag}(A))$
for $k = 1\colon n_b\colon n$
  % Allow for last incomplete block
  $jb = \min(n_b, n - k + 1)$
  % Store accumulated dot products
  $dots(k\colon n) = 0$
  if I own a diagonal block
    Send diag elements to process in my row
    and current column
  end
  if I am in the current column
    Receive diag blocks to *work*
  end
  % Current process col has all diag elements
  for $j = k\colon k + jb - 1$
    if I own elements in cols $k\colon k + jb - 1$
    compute my contribution of:
      if $j > k$
        $dots(i) = dots(i) + L(i, j-1)^2, \quad i = j\colon n$
      end
      $q = \min\Big\{p : L(p,p) - dots(p) = \max_{j\le i\le n}\{work(i,i) - dots(i)\}\Big\}$
      Broadcast pivot along process row
    else
      Receive pivot
    end
    if $L(q,q) \le tol$
      % computed rank of A is $j - 1$
      return
    end
    % global swaps
    swap $L(j,:)$ and $L(q,:)$
    swap $L(:,j)$ and $L(:,q)$
    swap $dots(j)$ and $dots(q)$
    swap $piv(j)$ and $piv(q)$
    if I own elements in cols $k\colon k + jb - 1$
    compute my contribution of:
      $L(j,j) = L(j,j) - dots(j)$
      $L(j,j) = \sqrt{L(j,j)}$
      % Update jth column
      if $1 < j < n$
        $L(j+1\colon n, j) = L(j+1\colon n, j) - L(j+1\colon n, 1\colon j-1)L(j, 1\colon j-1)^T$
      end
      if $j < n$
        $L(j+1\colon n, j) = L(j+1\colon n, j)/L(j,j)$
      end
    end
  end
  if $k + jb < n$
  % perform global block update
    $L(j+1\colon n, j+1\colon n) = L(j+1\colon n, j+1\colon n) - L(j+1\colon n, 1\colon j)L(j+1\colon n, 1\colon j)^T$
  end
end

5 Numerical Experiments

We timed and compared the following two routines:

- A double precision Fortran implementation of Algorithm 4.1, which we refer to hereafter as *our code*.
- The ScaLAPACK routine `PDPOTRF` which computes the unpivoted Cholesky factorization of a (double precision) positive definite matrix.

We generate a random symmetric positive definite matrix using the ScaLAPACK testing routine `PDMATGEN`. Note that as we are using positive definite matrices our code will not exit due to a pivot being less than or equal to the tolerance. This will enable us to see the pivoting overhead in our code. We do not look at the numerical behavior or rank detection properties of our code in this paper.

We compare the codes on three sizes of matrix, namely

$$n = 8000,\ 16000,\ 32000.$$

The following parameters are used and thus we tune the codes empirically. These were narrowed down after some initial runs.

- $NB = 16,\ 32,\ 64,\ 128,\ 256$
- PEs $= 4,\ 8,\ 16,\ 32,\ 64$ (XD1 only)
- Process grids of: $2\times 2,\ 2\times 4,\ 4\times 2,\ 2\times 8,\ 4\times 4,\ 8\times 2,\ 4\times 8,\ 8\times 4,\ 4\times 16,\ 8\times 8,\ 16\times 4$.

The tests were performed on the following machines:

- A Cray XD1 with 2.4 GHz Opteron processors and 2GB memory per processor and two processors per node.

- A Cray XT3 with 2.6 GHz Opteron processors and 2GB memory per processor.

In all cases we give the best time for each value of n and the number of processors. This was exclusively the grids with more rows than columns for both codes. A block size of 64 usually gave the quickest time for both codes.

Figure 3 shows the timings for the two routines on the XD1.

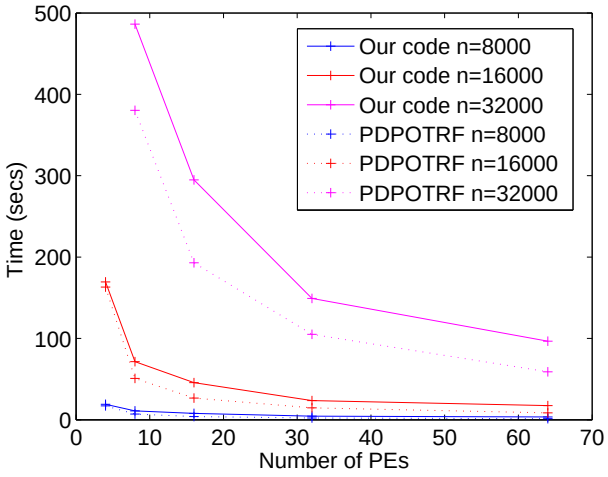

Figure 3: Timings for the XD1.

As we would expect with additional computation and communication the times to compute the pivoted factorization are longer. We show the pivoting overhead in Table 1. Here the figures show the difference in time of both codes as a percentage of the time for the unpivoted ScaLAPACK routine. We can see that as we increase the problem size that the overhead decreases. However, as we increase the number of processors the overhead generally increases. It is modest for runs on 4 processors but very large indeed for 64 processors.Thus we expect the code not to scale too well.

Table 1: Pivoting overhead on the XD1.

PEs	4	8	16	32	64
$n = 8000$	11.2	54.8	96.8	95.6	155.1
$n = 16000$	3.9	40.6	70.5	60.4	106.4
$n = 32000$		27.8	52.8	42.1	63.8

The scaling is shown in Figure 4 for $n = 8000$ and $n = 16000$. The green line shows linear scaling compared with the computation time of each routine on 4 processors. We see that the ScaLAPACK codes scales well to 64 processors, particulary for the larger problem size. Our code doesn't scale as well, although for $n = 16000$ we are scaling to 32 processors.

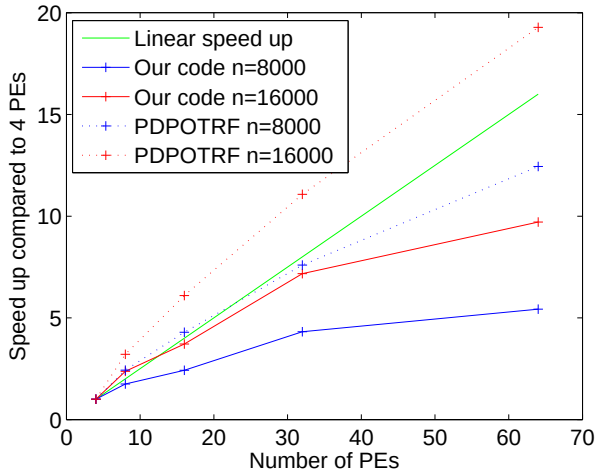

Figure 4: Scaling on the XD1.

As stated earlier we are only comparing the fastest times for each code over the range of block sizes and process grids. Figure 5 shoes the effect of choosing different values for a problem size of $n = 16000$ and on 64 processors. We can clearly see here that the optimal block size is 32 and optimal grid is 16-by-4. The block size effects whether whole blocks fit into cache for the matrix-vector and matrix-matrix operations as well as much of the GAXPY operations are done before the block update. With grids with more rows than columns, more processes are involved in the GAXPY operations and less involved in communicating the pivots.

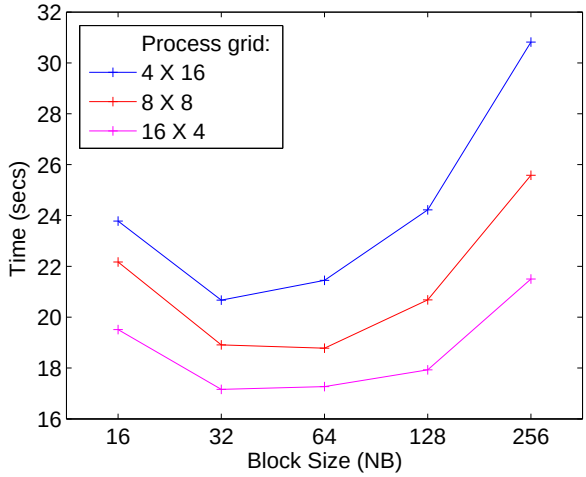

Figure 5: Effect of process grid and block size with $n = 16000$

Figure 6 show the timings for the XT3. We can see that the timings for larger problems and few processors are much quicker, up to 55% faster compared to the XD1.

However, as the numbers of processors increase the times are within 2 or 3%. This behavior means the codes doesn't scale quite as well on the XT3 as they did on the XD1.

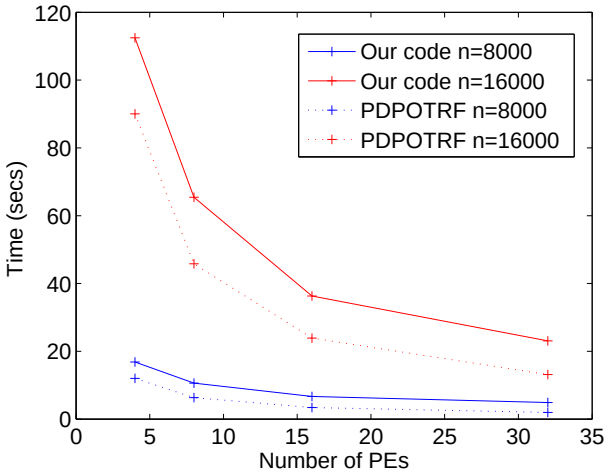

Figure 6: Timings for the XT3.

We can see in Figure 7 that on 32 processors the that a problem of $n = 8000$ gives a speed up of just over 3 times over the time on 4 processors, compared with approximately 4 on the XD1, and a speed up of nearly 5 times with $n = 16000$ compared to 7 times on the XD1.

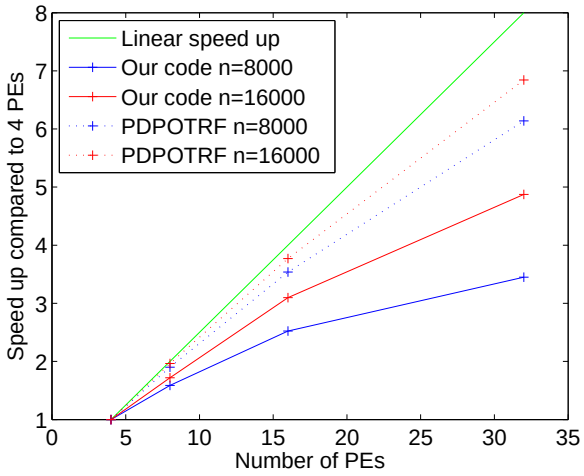

Figure 7: Scaling on the XT3.

6 Conclusions

The aim of the work in this paper was to ascertain the feasibility of implementing the Cholesky factorization with complete pivoting for ScaLAPACK. The code we demonstrated is still under development.

On smaller numbers of processes the pivoting only adds around 10% to the computation time. However, on higher numbers of processors this was as large as 155%. However, the overhead does decrease with problem size and larger problem sizes scaled to 32 processors.

We conclude that the code does appear to be practical but further investigation is required, particulary we would like to run larger problems on more processors, to see if the code will scale in these cases.

7 Acknowledgements

I would like to thank Aston University, and particularly Andrey Kaliazin, for providing access to an XD1. I would also like to thank The Swiss National Supercomputing Centre, and in particular Marie-Christine Sawley and Neil Stringfellow, for providing access to an XT3. Finally I would like to thank Kevin Roy at the University of Manchester for help in running the codes.

8 About the Author

Dr Craig Lucas

Craig has a PhD in Numerical Linear Algebra from the University of Manchester. He has worked in High Performance Computing at the university for the last two years. The HPC team provide a range of local and national services, including the national supercomputing service CSAR [10], with partners SGI and CSC.

Contact Information:

Manchester Computing
University of Manchester
Oxford Road Manchester
M13 9PL

Tel: +44 (0) 161 275 7029
Fax: +44 (0) 161 275 6800
Email: craig.lucas@manchester.ac.uk
Web: http://www.csar.cfs.ac.uk
http://www.mc.manchester.ac.uk/

References

[1] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Third edition, SIAM, Philadelphia, 1999.

[2] L. S. Blackford, J. Choi, A. Cleary, E. D'Azeuedo, J. Demmel, I. Dhillon, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK user's guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.

[3] J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, and R. C. Whaley. A proposal for a set of parallel basic linear algebra subprograms. LAPACK Working Note 100, May 1995.

[4] J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Soft.*, 16(1):1–17, 1990.

[5] J. Dongarra and R. C. Whaley. A user's guide to the BLACS v1.1. LAPACK Working Note 94, May 1997.

[6] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of basic linear algebra subprograms: Model implementation and test programs. *ACM Trans. Math. Soft.*, 14(1):18–32, 1988.

[7] G. H. Golub and C. F. Van Loan. *Matrix Computations*. Third edition, The Johns Hopkins University Press, Baltimore, MD, USA, 1996.

[8] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Second edition, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2002.

[9] Craig Lucas. LAPACK-style codes for level 2 and 3 pivoted Cholesky factorizations. LAPACK Working Note 161, February 2004.

[10] CSAR National Supercomputing Service. `http://www.csar.cfs.ac.uk/`.