Implementation of Out-of-Core Cholesky and QR Factorizations with POOCLAPACK*

Brian C. Gunter† Wesley C. Reiley‡ Robert A. van de Geijn§

September 3, 2000

Abstract

In this paper parallel implementation of out-of-core Cholesky factorization is used to introduce the Parallel Out-of-Core Linear Algebra Package (POOCLAPACK), a flexible infrastructure for parallel implementation of out-of-core linear algebra operations. POOCLAPACK builds on the Parallel Linear Algebra Package (PLAPACK) for in-core parallel dense linear algebra computation. Despite the extreme simplicity of POOCLAPACK, the out-of-core Cholesky factorization implementation is shown to achieve in excess of 80% of peak performance on a 64 node configuration of the Cray T3E-600. The insights gained from examining the Cholesky factorization have been applied to the much more difficult and important QR factorization operation. Preliminary results for parallel implementation of the resulting OOC QR factorization algorithm are included.

1 Introduction

There are only a few applications left that require the solution of extremely large dense linear systems. They tend to arise from boundary-element formulations for the solution of integral equations in the areas of electro-magnetics and acoustics [6, 8, 12]. Even for those applications, much cheaper methods based on multi-pole expansions, fast multipole methods (FMM), have recently become popular [11]. Nonetheless, there are still many such applications that are solved by forming large dense systems of equations. In some cases, this is simply because the users are naive. In other cases it is a conscious decision since a considerable effort is required to reformulate the problem in a fashion that allows fast multi-pole methods to be utilized. Furthermore, there are applications requiring the solution of large linear least squares problems that also give rise to very large linear systems [2]. For applications that do still lead to large dense linear systems, the matrices involved are frequently so large that they do not fit even in the combined memories of the processors of a large distributed memory parallel supercomputer. Such problems are often referred to as

*This project was partially funded by NASA as part of the Gravity Recovery and Climate Experiment (GRACE), NAS5-97213.
†Center for Space Research, The University of Texas, Austin, TX 78712, gunter@csr.utexas.edu
‡Department of Computer Sciences, The University of Texas, Austin, TX 78712, wesley@cs.utexas.edu
§Department of Computer Sciences, The University of Texas, Austin, TX 78712, rvdg@cs.utexas.edu

out-of-core problems, since they do not fit in the core memory of the computer. The matrices are instead stored on disk.

The preeminent library for sequential computers and conventional (shared memory) vector supercomputers is the Linear Algebra Package (LAPACK) [1]. This package does not explicitly include out-of-core capabilities, although on machines with virtual memory the library can be used to solve problems larger than fit in-core. For larger problems, a version of this library called ScaLAPACK [5], designed for distributed memory parallel architectures, can be used. This extension of LAPACK does include prototype out-of-core implementations of some of the ScaLAPACK routines, including general linear solvers via LU factorization, positive definite linear solvers via Cholesky factorization, and linear least squares solvers via QR factorization [7]. However, the ScaLAPACK out-of-core approach blocks matrices to be brought in-core by "slabs" of columns. This approach is inherently nonscalable since there is a fixed amount of aggregate memory among p processors, as larger and larger matrices are factored, the panel of columns that fits in-core becomes more narrow, inherently affecting both the performance of the in-core kernels as well as the ratio of computation to I/O operations.

A more serious effort to add out-of-core capabilities to LAPACK and ScaLAPACK is provided by SOLAR [20], a portable library for scalable out-of-core linear algebra computations. This library uses ScaLAPACK routines for in-core computation, but provides an I/O layer that manages matrix input-output. SOLAR achieves better I/O rates by allowing a different storage scheme for matrices on disk than is used in-core by ScaLAPACK. Impressive performance is reported for up to four nodes of an IBM SP-2. Note that while their approach to Cholesky is somewhat similar to ours, they do not extend the approach to the LU or QR factorization, for which they use a slab approach similar to that used by ScaLAPACK.

Our own approach is somewhat different. Since we developed the Parallel Linear Algebra Package (PLAPACK) [21] used as a basis for the Parallel Out-of-Core Linear Algebra Package (POOCLAPACK), we have more flexibility to customize both the in-core and the out-of-core algorithms. This in turn allows us to code the out-of-core algorithms in such a way that the I/O of matrices becomes trivial, reducing the amount of code required to port between platforms and improving performance. Furthermore, it allows us to create in-core kernels that allow novel out-of-core approaches to be implemented, as will become apparent when we discuss out-of-core QR factorization.

It should be noted that the above described parallel out-of-core library efforts are in addition to a number of parallel out-of-core implementations of individual operations or machine specific libraries for dense linear systems reported in the literature [2, 14, 4, 18, 19]. Additional references to applications requiring large dense linear solves are given in [6, 8, 12]. Additional references to research using fast summation methods like FMM are given in [11].

This paper is organized as follows: Section 2 discusses issues regarding the in-core and out-of-core implementation of sequential Cholesky factorization. Section 3 we briefly discuss how the techniques can be extended to the QR factorization, requiring in-core and OOC algorithms that are not supported by LAPACK, ScaLAPACK, or SOLAR. Section 4 introduces the POOCLAPACK approach to coding parallel out-of-core dense linear algebra algorithms. Performance is reported in Section 5. Concluding remarks and future directions are given in the final section.

2 Sequential Implementation

Two algorithms often used for blocked Cholesky factorization, known as the right- and left-looking variant [9], are given in Fig. 1. For details of how to derive the algorithm we refer the reader to [16, 17].

partition $A = \left(\begin{array}{c|c} A_{TL} & \star \\ \hline A_{BL} & A_{BR} \end{array} \right)$ where A_{TL} is 0×0

do until A_{BR} is 0×0

repartition

$$\left(\begin{array}{c||c} A_{TL} & \star \\ \hline\hline A_{BL} & A_{BR} \end{array} \right) = \left(\begin{array}{c||c|c} A_{00} & \star & \star \\ \hline\hline A_{10} & A_{11} & \star \\ \hline A_{20} & A_{21} & A_{22} \end{array} \right) \text{ where } A_{TL} \text{ is } b \times b$$

right-looking algorithm	left-looking algorithm
$A_{11} \leftarrow L_{11} = \text{Chol.fact.}(A_{11})$	$A_{11} \leftarrow A_{11} - A_{10} A_{10}^T$
$A_{21} \leftarrow L_{21} = A_{21} L_{11}^{-T}$	$A_{21} \leftarrow A_{21} - A_{20} A_{10}^T$
$A_{22} \leftarrow A_{22} - L_{21} L_{21}^T$	$A_{11} \leftarrow L_{11} = \text{Chol.fact.}(A_{11})$
	$A_{21} \leftarrow L_{21} = A_{21} L_{11}^{-T}$

continue with

$$\left(\begin{array}{c||c} A_{TL} & \star \\ \hline\hline A_{BL} & A_{BR} \end{array} \right) = \left(\begin{array}{c|c||c} A_{00} & \star & \star \\ \hline A_{10} & A_{11} & \star \\ \hline\hline A_{20} & A_{21} & A_{22} \end{array} \right)$$

enddo

Figure 1: Blocked right- and left-looking Cholesky factorization algorithms.

Either of these two algorithms can be used for efficient sequential in-core implementation of the Cholesky factorization. In practice, the right-looking algorithm is favored for reasons that go beyond the scope of this paper.

The left-looking Cholesky factorization is favored for out-of-core implementations. There are two basic reasons for this: First, the left-looking Cholesky requires approximately half the I/O operations of the right-looking algorithm. Second, it is easier to add *check-pointing* to a left-looking algorithm. Check-pointing allows for a restart partially into the computation in case of a system failure.

Let us examine in more detail how to implement an out-of-core Cholesky factorization. Partition

$$A = \left(\begin{array}{c||c|c} L_{00} & * & * \\ \hline\hline L_{10} & A_{11} & * \\ \hline L_{20} & A_{21} & A_{22} \end{array} \right)$$

where L_{00} is $m \times m$ and we assume that L_{*0} have been computed, while the other parts of A have been left untouched. Here A_{11} is of size $t \times t$, which we will later call a *tile* of size t. All data is assumed to exist on disk.

The following steps will advance the computation so that L_{11} and L_{21} have been computed and have overwritten the corresponding blocks of A:

1. Read A_{11} from disk into memory.
2. Update $A_{11} \leftarrow A_{11} - L_{10}L_{10}^T$ where L_{10} is on disk.
3. Update $A_{11} \leftarrow L_{11} = \text{Chol.fact.}(A_{11})$. Since A_{11} is in memory, this requires an in-core Cholesky factorization. As mentioned, typically a right-looking variant is favored for this subproblem.
4. Write L_{11} to disk, leaving a copy in memory.
5. Update $A_{21} \leftarrow (A_{21} - L_{20}L_{10}^T)L_{11}^{-T}$, where A_{21}, L_{20} and L_{10} are on disk and L_{11} is in memory.
6. Flush all memory.

We must give further details on how to perform steps 2 and 5:

Step 2: $A_{11} \leftarrow A_{11} - L_{10}L_{10}^T$: Here A_{11} is in memory, but L_{10} is on disk. At first glance, this appears to require a read of L_{10}, followed by an in-core symmetric rank-k update. This requires $t \times m$ data to be read, after which mt^2 floating point operations are performed to update A_{11}, for a ratio of t floating point operations for every floating point number read. However, reading L_{10} requires a considerable amount of memory, thereby limiting the size of t, and thus affecting this ratio.

Key observation: The following approach retains the benefits of the same ratio t of computation to disk accesses, while maximizing the size of t and thus this ratio: Partition $L_{10} = \left(L_{10}^{(0)} \,\middle|\, \cdots \,\middle|\, L_{10}^{(k-1)} \right)$ where $L_{10}^{(j)}$ has approximately b columns. Notice that

$$A_{11} - L_{10}L_{10}^T = A_{11} - L_{10}^{(0)}L_{10}^{(0)\,T} - \cdots - L_{10}^{(k-1)}L_{10}^{(k-1)\,T}$$

Thus, the following procedure will perform the update of A_{11}. For each $L_{10}^{(j)}$, read this submatrix ($t \times b$ items read), and perform an in-core rank-k update (bt^2 floating point operations). Notice that this maintains the ratio of t computations for each item read from disk. However, by picking b relatively small, very little memory is needed for L_{10}, thus allowing t to be chosen to be much larger. The block size b is typically chosen to equal a block-size that maximizes the performance of the in-core symmetric rank-k update.

This "sequence of narrow symmetric rank-k updates" approach to implementating a larger symmetric rank-k update yields an excellent parallel in-core implementation of symmetric rank-k update. Thus, the out-of-core approach fits naturally with a very good in-core algorithm[1].

Step 5: $A_{21} \leftarrow L_{21} = (A_{21} - L_{20}L_{10}^T)L_{11}^{-T}$: Here only L_{11} is in memory. In [17] we show that the exact same technique as described above for Step 2 can be used to allow almost all in-core memory to be used to store L_{21}, with similar benefits. Again, the approach meshes will with how are naturally implementedparallel in-core implementation of matrix-matrix multiplication [21, 13].

[1]Unfortunately, the only reference for this is the actually implementation of symmetric rank-k update in the PLAPACK source. Most likely, ScaLAPACK uses a similar approach.

Key observation: Careful consideration of the complete out-of-core algorithm shows that in addition to two tiles of size $t \times t$ (one for A_{11} and for $A_{21}^{(i)}$) only a small amount of workspace is needed for storing a few blocks of columns of L_{10} and $L_{20}^{(i)}$. Naturally, t is chosen as large as possible, thus improving the ratio of computation to disk accesses. In [16] we show how this can be brought down to just one tile. However, the two-tile approach meshes better with how an OOC QR factorization is naturally implemented.

Two techniques allow for further reduction in I/O overhead: It is possible to exploit asynchronous I/O operations to overlap computation with I/O operations. By storing matrices on disk we arrange for all reads from disk to access very large contiguous blocks of data. For details, see [17].

3 Out-of-Core QR factorization

As part of an effort to compute the parameters of the Earth's gravitational field, POOCLAPACK is being used to solve a very large linear least-squares problem. The below described out-of-core QR factorization is one of several dense linear algebra algorithms being developed in support of that effort.

Given an $m \times n$ matrix A, its QR factorization is given by $A = QR$ where Q is unitary and R is upper-triangular. For simplicity, we will assume that $m >= n$. Matrix Q is usually computed and stored as a collection of Householder transformations. A blocked algorithm is derived by aggregating a number of Householder transforms into a WY-transform [3] or YTY^T-transform [10].

The primary problem with creating an OOC version of the QR factorization is that on the surface it appears that columns of matrix A must be brought into memory simultaneously in order to compute Householder transforms from columns or apply Householder transforms to columns. In our approach we break this dependence as follows: Partition

$$A = \left(\begin{array}{c|c|c} A_{11} & \cdots & A_{1N} \\ \hline A_{21} & \cdots & A_{2N} \\ \hline \vdots & \cdots & \vdots \\ \hline A_{M1} & \cdots & A_{MN} \end{array} \right)$$

where A_{ii} is square. A sketch of an OOC QR factorization is given by the followgin:

- for $j = 1 : N$
  - Compute the QR factorization $A_{jj} = Q_{jj} R_{jj}$ leaving Q_{jj} in compact form (storing the Householder vectors below the diagonal of A_{jj}.
  - Update $A_{jk} = Q_{jj}^T A_{jk}$ by applying the Householder transforms stored in A_{jj}.
  - for $i = j + 1 : M$
    * Compute $\left(\frac{R_{jj}}{A_{ij}} \right) \rightarrow Q_{ij} \left(\frac{\hat{R}_{jj}}{0} \right)$ overwriting R_{jj} with $\hat{R_{jj}}$.

      Key insight: The Householder vector that zeroes entries below the (p, p) element of R_{jj} has the form $\left(\frac{e_p}{v_p} \right)$, a special structure that can be exploited when applying the Householder vector to a matrix, when building a WY-transform or YTY^T-transform, and when storing the Householder vector.

    * for k=j+1:N
        · Update $\left(\frac{A_{jk}}{A_{ik}}\right) \leftarrow Q_{ij}^T \left(\frac{A_{jk}}{A_{ik}}\right)$.

          Key insight: Again, one can take advantage of the special form of the Householder transforms.
    * endfor
  - endfor
- endfor

Details of how the specialized operation $\left(\frac{R_{jj}}{A_{ij}}\right) \rightarrow Q_{ij} \left(\frac{\hat{R}_{jj}}{0}\right)$ is implemented can be found in [15]. Details on how to implement the other specialized kernels go beyond the scope of this extended abstract.

4 Parallel Implementation

The Parallel Linear Algebra Package (PLAPACK) is a flexible infrastructure for implementing parallel dense linear algebra libraries. An MPI-like programming interface, which hides details about matrices and vectors like distribution from the user, makes both the library implementation and its use considerably simpler than more conventional packages like ScaLAPACK. In addition, the simple programming approach allows more complex algorithms to be implemented, which often yield better performance.

The discriptions of the out-of-core sequential Cholesky and QR factorizations translate directly to POOCLAPACK code. To illustrate the simplicity of the code we include POOCLAPACK code for OOC Cholesky factorization in Fig. 2.

5 Performance

In this section, we report preliminary performance achieved with the described PLAPACK based parallel out-of-core implementations of the Cholesky and QR factorizations.

We demonstrate performance on the Cray T3E-600 (300 MHz) with all computations performed in 64-bit arithmetic. The algorithms were implemented using an alpha release of PLAPACK Version R2.03, which performs all communication by means of MPI. We report performance measuring MFLOP/s/processor (millions of floating point operations per second per processor). For reference, the matrix-matrix multiplication on a single processor of the T3E-600 in MFLOP/s attains up to 445 MFLOP/s. All performance reported in this section was measured with data streams turned on (a hardware feature that adds about 15–20% to the performance of the local matrix-matrix multiply kernel).

In [17, 16] we report performance of a number of different implementations of the Cholesky factorization including versions that did and did not overlap I/O with computation and versions that did and did not force all I/O to be in large contiguous blocks. Here we report performance only for the in-core PLAPACK Cholesky factorization (`In-core Chol` in the table) and a version of the OOC Cholesky factorization

```
int POOCLA_Chol_by_panels( int N, PLA_Obj *A_row_panels )
{
  < declarations >

  size_done = 0;                              /* number of columns finished */
  for ( j=0; j<N; j++ ){
    PLA_Obj_global_length( A_row_panels[ j ], &t );     /* get tile size    */

    /* View current L_10 and A_11 submatrices */
    PLA_Obj_vert_split_2( A_row_panels[ j ], size_done,  &L_10, &temp );
    PLA_Obj_vert_split_2( temp,              t,          &A_11, PLA_DUMMY );

    /* Create an in-core matrix into which to copy A_11 */
    PLA_Matrix_create_conf_to( A_11, &A_11_in );
    PLA_Copy( A_11, A_11_in );

    /* Update A_11 <- A_11 - L_10 * L_10, A_11 in-core, L_10 out-of-core */
    POOCLA_Syrk( PLA_LOWER_TRIANG, PLA_NO_TRANS, min_one, L_10, one, A_11_in );

    /* Factor updated in-core A_11 and write out the result */
    PLA_Chol( PLA_LOWER_TRIANGULAR, A_11_in );
    PLA_Copy( A_11_in, A_11 );

    /* Loop over A_21^i */
    for ( i=j+1; i<N; i++ ){
      /* View current matrices L_20^i and A_21^i */
      PLA_Obj_vert_split_2( A_row_panels[ i ], size_done, &L_20_1, &temp );
      PLA_Obj_vert_split_2( temp,              t, &A_21_1, PLA_DUMMY );

      /* Create an in-core matrix into which to copy A_21^i */
      PLA_Matrix_create_conf_to( A_21_1, &A_21_1_in );
      PLA_Copy( A_21_1, A_21_1_in );

      /* Update A_21^i <- A_21^i - L_20 * L_10^T */
      POOCLA_Gemm( PLA_NO_TRANS, PLA_TRANS,
                   min_one, L_20_1, L_10, one, A_21_1_in );

      /* Update A_21^i <- L_21^i = A_21^i * L_11^-T */
      PLA_Trsm( PLA_SIDE_RIGHT, PLA_LOWER_TRIANG,
                PLA_TRANS, PLA_NONUNIT_DIAG,
                one, A_11_in, A_21_1_in );

      /* Write out A_21^i */
      PLA_Copy(A_21_1_in, A_21_1);

      size_done += t;
    }
  }
  < clean up >
}
```

Figure 2: POOCLAPACK Out-of-Core Cholesky factorization. In this version, the matrix is presented as a collection of panels of rows in an effort to improve disk performance.

that does NOT use asynchronous I/O but DOES use a specialized storage scheme that allows for large contiguous reads from disk (`Chol_by_panels` in the table). We report performance from two different Cray T3E systems: the first a smaller system at UT-Austin (up to 80 compute nodes) and the second a now decommissioned machine at the Goddard SFC. The primary reason is to show the degradation in performance observed when executing on our local machine. This degradation helps explain the degradation of the performance of the QR factorization, for which we at this time do not have results on a machine with a more reasonable I/O setup.

For a fixed number of processors, we report performance for a problem equal to the tile size $t \times t$, $(2t)\times(2t)$, and $(3t)\times(3t)$. For those familiar with PLAPACK, a distribution block size of 24 and algorithmic block size of 128 was used for the Cholesky factorization. The block size described in Section 2 used for partitioning L_{10} and $L_{20}^{(i)}$, b, was taken to equal the algorithmic block size.

The Cray T3E Systems have an extended IO system, called Flexible File IO (FFIO). This system allows the user to insert layers through which data is passed. Within the layer, the user can insert various kinds of buffers and caches. Cache and/or buffer sizes and properties like striping across multiple disks can be controlled by command line routines. We experimented with putting a small cache between disk and memory and used default striping settings. It should be noted that changes in the configuration of the files and cache sizes did not seem to affect performance of our algorithms much on the Goddard SFC Cray T3E, which has much more impressive I/O capabilities. In particular, the more sophisticated algorithms that allowed larger blocks of contiguous data to be read did not seem to be affected at all on that machine.

Performance of the QR factorization is very preliminary. Notice that performance is impressive for up to 16 processors, but degrades considerably when 64 processors are utilized. It is our believe that this is due to the I/O limitations of the Cray T3E-600 at UT-Austin. Furthermore, we have yet to determine optimal blocking sizes nor did we experiment with FFIO for the OOC QR factorization.

POOCLAPACK has been successfully ported to a wide range of platforms (essentially all platforms that already support PLAPACK). A more complete set of performance numbers, including performance on additional platforms, is planned for the final paper.

6 Conclusion

We have described a simple extension to the PLAPACK parallel linear algebra infrastructure that allows for elegant yet high-performance implementation of out-of-core dense linear algebra algorithms. Since both PLAPACK and its out-of-core extension provide a simple abstract programming interface, the implementations lend themselves to customization to allow new functionality to be added as is demonstrated for the QR factorization.

More information

For more information on PLAPACK and POOCLAPACK visit

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

Algorithm	p	tile size t	1×1 tiles ($n = t$)			2×2 tiles ($n = 2t$)			3×3 tiles ($n = 3t$)		
			MFLOP/s /proc.	Time (sec) Total	Time (sec) I/O	MFLOP/s /proc.	Time (sec) Total	Time (sec) I/O	MFLOP/s /proc.	Time (sec) Total	Time (sec) I/O
Cholesky Factorization on UT-Austin T3E-600											
`In-core Chol`	1	2088	258								
`Chol_by_panel`	1	2088	254	11.9	0.39	279	88	2	343	239	5
`In-core Chol`	4	4704	312								
`Chol_by_panel`	4	4704	290	29.9	1.9	346	201	7	366	639	20
`In-core Chol`	16	8448	313								
`Chol_by_panel`	16	8448	273	46.1	5.8	327	306	21	349	970	50
`In-core Chol`	64	18432	318								
`Chol_by_panel`	64	18432	249	132	29.5	309	843	98	316	2786	306
Cholesky Factorization on GSFC T3E-600											
`In-core Chol`	1	2088	263	11.5							
`Chol_by_panel`	1	2088	245	12.4	1.0	296	82	9	334	245	17
`In-core Chol`	4	4704	304	28.5							
`Chol_by_panel`	4	4704	276	31.5	2.6	331	209	10	353	663	24
`In-core Chol`	16	8448	304	41.3							
`Chol_by_panel`	16	8448	273	46.1	4.3	321	313	13	343	989	32
`In-core Chol`	64	18432	263	124							
`Chol_by_panel`	64	18432	267	122	15.0	315	827	53	331	2654	125
QR Factorization on UT-Austin T3E-600 (preliminary results)											
`QR_by_panel`	1	2048	242	47.3		261	351		256	1206	
`QR_by_panel`	4	4096	292	78.6		293	626		303	2038	
`QR_by_panel`	16	8192	288	158.9		269	1361		273	4534	
`QR_by_panel`	64	16384	213	430.8		202	3635		207	11896	

Table 1: Performance of the Cholesky factorization on the Cray T3E-600 at UT-Austin.

Acknowledgments

Access to equipment for development of the described infrastructure was provided by the National Partnership for Advanced Computational Infrastructure (NPACI) and The University of Texas Advanced Computing Center (TACC). We also gratefully acknowledge access to the Cray T3E-600 System at the Goddard Space Flight Center provided by the NASA HPCC Earth and Space Science Project.

References

[1] E. Anderson, Z. Bai, J. Demmel, J. E. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. E. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, 1992.

[2] Gregory A. Baker. *Implementation of Parallel Processing to Selected Problems in Satellite Geodesy*. PhD thesis, The University of Texas at Austin, 1998.

[3] Christian Bischof and Charles Van Loan. The WY representation for products of Householder matrices. *SIAM J. Sci. Stat. Comput.*, 8(1):s2–s13, Jan. 1987.

[4] Jean-Philippe Brunet, Palle Pederson, and S. Lennart Johnsson. Load-balanced LU and QR factor and solve routines for scalable processors with scalable I/O. In *Proceedings of the 17th IMACS World Congress*, Atlanta, Georgia, July 1994.

[5] 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.

[6] 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.

[7] E. F. D'Azevedo and J. J. Dongarra. The design and implementation of the parallel out-of-core scalapack lu, qr, and cholesky factorization routines. LAPACK Working Note 118 CS-97-247, University of Tennessee, Knoxville, Jan. 1997.

[8] 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.

[9] Jack J. Dongarra, Iain S. Duff, Danny C. Sorensen, and Henk A. van der Vorst. *Solving Linear Systems on Vector and Shared Memory Computers*. SIAM, Philadelphia, PA, 1991.

[10] Jack J. Dongarra, Sven J. Hammarling, and Danny C. Sorensen. Block reduction of matrices to condensed forms for eigenvalue computations. *Journal of Computational and Applied Mathematics*, 27, 1989.

[11] Y. Fu, K. J. Klimkowski, G. J. Rodin, E. Berger, J. C. Browne, J. K. Singer, R. A. van de Geijn, and K. S. Vemaganti. A fast solution method for three-dimensional many-particle problems of linear elasticity. *Int. J. Num. Meth. Engrg.*, 42:1215–1229, 1998.

[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] John Gunnels, Calvin Lin, Greg Morrow, and Robert van de Geijn. A flexible class of parallel matrix multiplication algorithms. In *Proceedings of First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing (1998 IPPS/SPDP '98)*, pages 110–116, 1998.

[14] Ken Klimkowski and Robert van de Geijn. Anatomy of an out-of-core dense linear solver. In *Proceedings of the International Conference on Parallel Processing 1995*, volume III - Algorithms and Applications, pages 29–33, 1995.

[15] Enrique S. Quintana-Orti and Robert van de Geijn. Fast parallel kernels for selected problems in control theory. In *Proceedings of the SIAM Conference on Parallel Processing for Scientific Computing 1999*, 1999.

[16] Wesley C. Reiley. Efficient parallel out-of-core implementation of the cholesky factorization". Technical Report CS-TR-99-33, Department of Computer Sciences, The University of Texas at Austin, Dec. 1999. Undergraduate Honors Thesis.

[17] Wesley C. Reiley and Robert A. van de Geijn. Pooclapack: Parallel out-of-core linear algebra package. Technical Report CS-TR-99-33, Department of Computer Sciences, The University of Texas at Austin, Nov. 1999.

[18] David S. Scott. Out of core dense solvers on Intel parallel supercomputers. In *Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation*, pages 484–487, 1992.

[19] David S. Scott. Parallel I/O and solving out-of-core systems of linear equations. In *Proceedings of the 1993 DAGS/PC Symposium*, pages 123–130, Hanover, NH, June 1993. Dartmouth Institute for Advanced Graduate Studies.

[20] Sivan Toledo and Fred G. Gustavson. The design and implementation of SOLAR, a portable library for scalable out-of-core linear algebra computation. In *Proceedings of IOPADS '96*, 1996.

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