THE DESIGN OF SCALABLE SOFTWARE LIBRARIES
FOR DISTRIBUTED MEMORY CONCURRENT COMPUTERS

Jaeyoung Choi

University of Tennessee
Department of Computer Science
Knoxville, TN 37996-1301

Jack Dongarra*

David W. Walker

Oak Ridge National Laboratory
Mathematical Sciences Section
Computer Science and Mathematics Division
Oak Ridge, TN 37831-6367

*Also with the Department of Computer Science
University of Tennessee
107 Ayres Hall
Knoxville, TN 37996-1301

"The submitted manuscript has been authored by a contractor of the U.S. Government under contract No. DE-AC05-84OR21400. 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."

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

* This work was supported in part by DARPA and ARO under contract number DAAL03-91-C-0047, and in part by the Applied Mathematical Sciences Research Program, Office of Energy Research, U.S. Department of Energy under contract DE-AC05-84OR21400 with the Martin Marietta Energy Systems, Inc.

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

The Design of Scalable Software Libraries for Distributed Memory Concurrent Computers

Jaeyoung Choi[†], Jack J. Dongarra^{†‡}, and David W. Walker[‡]

[†]Oak Ridge National Laboratory
Mathematical Sciences Section
P. O. Box 2008
Oak Ridge, TN 37831-6367

[‡]University of Tennessee
Department of Computer Science
107 Ayres Hall
Knoxville, TN 37996-1301

Abstract

This paper describes the design of ScaLAPACK, a scalable software library for performing dense and banded linear algebra computations on distributed memory concurrent computers. The specification of the data distribution has important consequences for interprocessor communication and load balance, and hence is a major factor in determining performance and scalability of the library routines. The block cyclic data distribution is adopted as a simple, yet general-purpose, way of decomposing block-partitioned matrices. Distributed memory versions of the Level 3 BLAS provide an easy and convenient way of implementing the ScaLAPACK routines.

1 INTRODUCTION

Massively parallel, distributed memory, concurrent computers are playing an increasingly important role in large scale scientific computing, and in particular are the primary target machines for "Grand Challenge" applications. In addition, massively parallel computers are also beginning to be more widely used in production engineering environments, and to a somewhat lesser extent in the business and finance sectors. A number of key software technologies are helping to accelerate the more widespread use of massively parallel computers in these areas. These include the development of parallelizing compilers, parallel language extensions such as Fortran D [1] and High Performance Fortran, support for parallel constructs and operations, such as guard layers and object migration, and a variety of tools for debugging, and visualizing/analyzing performance on massively parallel computers.

Another important and active research area is the development of re-usable software for multicomputers in the form of libraries and "tool-kits" [2, 3, 4]. We are currently building a software library for performing dense and banded matrix computations on multicomputers. This library, named ScaLAPACK, will be fully compatible with the LAPACK library for vector and shared memory computers [5, 6, 7, 8] and is designed to be used within Fortran D and High Performance Fortran programs. ScaLAPACK is based on parallel algorithms that are scalable on current multicomputers, such as the Intel Delta and Paragon machines and the CM-5, and on future machines of similar design. The next section discusses what is meant by scalability in the context of our work. We expect that the same techniques used to ensure scalability on these machines will also help in implementing the library efficiently on machines with logically shared memory, such as the KSR-1 and Tera computers.

Much of the work in implementing ScaLAPACK has involved the consolidation of previous work on parallel dense linear algebra algorithms, although some new algorithmic issues have arisen in the course of the work. Most of the new ideas incorporated into ScaLAPACK are concerned with general design issues directed at ensuring that the library is robust, general-purpose, flexible, scalable, and easy to use. These issues will be discussed in Section 3.

In Section 4 we investigate the performance of a right-looking variant of the LU factorization algorithm on the Intel Delta multicomputer. In particular, we show how the performance of this ScaLAPACK routine depends on the data decomposition, and demonstrate its scalability.

We expect ScaLAPACK to be used in computations that use the boundary element method, such as the two-dimensional elastodynamics problem men-

tioned in Section 5, and in large-scale electromagnetic scattering problems. In addition, some of the routines will serve as building blocks for libraries for sparse solvers that are also currently under development.

2 SCALABILITY

According to Gupta and Kumar [9] a parallel algorithm is scalable if the concurrent efficiency can be held constant as the number of processors increases by increasing the problem size. The concurrent efficiency is defined by

$$\epsilon(N, N_p) = \frac{1}{N_p} \frac{T_{\text{seq}}(N)}{T(N, N_p)}, \quad (1)$$

N_p is the number of processors, N is the problem size, $T(N, N_p)$ is the time to solve a problem of size N on N_p processors, and $T_{\text{seq}}(N)$ is the time to solve the same problem on a single processor using the best sequential algorithm. Many parallel algorithms, particularly dense linear algebra algorithms, are parallelized versions of the best sequential algorithm, so that floating point operations performed in the parallel and sequential codes are the same. Assuming the floating point operation count dominates the performance of the sequential code, then the performance of the parallel code can be expressed as

$$G(N, N_p) = \frac{N_p}{t_{\text{calc}}} \epsilon(N, N_p) \quad (2)$$

where t_{calc} is the time to perform a floating point operation on a processor. Thus, the performance per processor is just proportional to the concurrent efficiency.

In order to give a clear idea of what we mean by “scalability” we will now introduce the isoefficiency function, $\rho_\epsilon(N_p)$, which is defined to be the problem size necessary to maintain some fixed efficiency, ϵ , as the number of processors, N_p , varies. Thus, if $\rho_\epsilon(N_p)$ depends linearly on N_p then the concurrent efficiency is a function of the grain size, $g = N/N_p$. Such algorithms are highly scalable. Algorithms for which $\rho_\epsilon(N_p)$ is a rapidly increasing function of N_p are said to scale poorly.

The scalability of a parallel algorithm can be assessed by plotting the isoefficiency function for different values of ϵ , that is by plotting curves in the (N_p, N) plane on which ϵ is constant. On any particular machine we are usually interested how an algorithm scales within a “window” in the (N_p, N) plane. This is shown schematically in Figure 1 in which the window

Figure 1: Schematic representation of isoefficiency curves within the window of interest, shown unshaded. The solid curves represent isoefficiency curves labeled by the efficiency. The dashed lines correspond to high scalability.

is bounded to the right by the number of processors in the parallel machine, below by the size of smallest problem of interest, and to the left and above by either the memory size per processor or runtime considerations. For some algorithms there may also be an upper bound on the problem size imposed by considerations of stability and/or accuracy. In Figure 1 we have assumed that the memory requirements scale linearly with the problem size. The runtime bound turns over as N_p increases as the concurrent efficiency falls off.

In designing an algorithm for a scalable library we seek one that is scalable within the windows of interest of as large a set of machines as possible. It should be noted that scalability studies conducted on small machines are of little use if the problem sizes considered are below the minimum size of interest. In the absence of an accurate performance model it is not valid to infer that an algorithm that scales well on one machine will also scale well on another machine, or even on the same machine for a larger number of processors. It is also important to note the distinction between good scalability and high efficiency. Suppose we have two algorithms with the same order of computational complexity that perform the same task. Algorithm 1 is efficient and scalable for some number of processors $N_p \leq N_*$, but loses scalability as the number of processors increases further. Algorithm 2 is less efficient than Algorithm 1 for a given problem size in the $N_p \leq N_*$ regime, but maintains good scalability for all N_p . In this case it is unclear which algorithm is “best” overall. For a sufficiently small number of

processors Algorithm 1 is preferable, but for a larger machine Algorithm 2 is better. Such dilemmas raise the possibility of using polyalgorithms, that is applying different algorithms on different machines or for different regions of the (N_p, N) plane.

3 DESIGN ISSUES

The main design goals of the ScaLAPACK library are that it should be robust, general purpose, flexible, scalable, and easy to use on a broad range of MIMD distributed memory concurrent computers. In general, scalability is enhanced by ensuring good load balance, and by minimizing interprocessor communication costs. This latter goal can be achieved by reducing the number of messages sent, reducing the message volume, and by overlapping communication and computation whenever possible. In many cases there are tradeoffs between load imbalance and communication costs, and these tradeoffs, and hence the performance and scalability of a parallel code, depend on the layout of an application's data within the hierarchical memory of the concurrent computer. On shared memory concurrent computers the software package LAPACK seeks to make efficient use of the hierarchical memory by maximizing data reuse, i.e., on a cache-based computer by avoiding having to reload the cache too frequently. LAPACK does this by casting linear algebra computations in terms of block-oriented, matrix-matrix operations known as the Level 3 BLAS [10] whenever possible. This approach generally results in maximizing the ratio of floating point operations to memory references, and reuses data as much as possible while it is stored in the highest levels of the memory hierarchy (for example, vector registers, or high-speed cache).

An analogous approach has been followed in the design of ScaLAPACK for distributed memory machines. By using block-partitioned algorithms we seek to reduce the frequency with which data must be transferred between processors, thereby reducing the fixed startup cost (or latency) incurred each time a message is communicated. An example of a block-partitioned algorithm is given in Section 4.

In decomposing a block-partitioned matrix over the processors of a concurrent computer it is natural to distribute the blocks as indivisible units. In general, a vector of data items can be decomposed by mapping the processors to a linear sequence and assigning the item with global index m to the p th processor in the sequence, where it is stored in a vector with local index i . This can be expressed by means of a

mapping $\mu(m) = (p, i)$. The decomposition of a block-partitioned matrix can be regarded as the tensor product of two such mappings, μ and ν . The processors are arranged as a logical two-dimensional mesh, or template, of $N_p = P \times Q$ processors. If

$$\mu(m) = (p, i), \quad \nu(n) = (q, j) \quad (3)$$

then the block with global index (m, n) is assigned to the processor at location (p, q) of the template, where it is stored at location (i, j) in a local block-partitioned matrix. Thus, μ and ν decompose the row index set and the column index set, respectively.

Linear algebra computations often involve the elimination of rows and/or columns as the algorithm progresses. To ensure good load balance in such cases we use the cyclic decomposition in which the data items assigned to a processor are separated by a fixed stride in the vector to be decomposed. Thus, we restrict ourselves to mappings of the form

$$\sigma(\ell) = (\ell \bmod R, \lfloor \ell/R \rfloor) \quad (4)$$

where R is the number of processors over which the vector is decomposed (P in the case of μ , and Q in the case of ν). This way of decomposing a block-partitioned matrix is referred to as a *block cyclic decomposition*. Examples are given in Figure 2.

The block cyclic decomposition is parameterized by 4 quantities, r , s , P , and Q , and a wide range of decompositions can be achieved by varying these parameters. However, in some cases nonsquare blocks can result in more complicated parallel algorithms, and give rise to additional sources of overhead. For example, in the LU factorization algorithm discussed in Section 4, if $r > s$ then the panel to be factored in each stage of the algorithm is spread over two columns of processors in the template. But if $r < s$ then performing the triangular solve would, in general, involve two rows of processors in the template. Only if the blocks are square, i.e., $r = s$, does the factorization involve just one column of the processor template, and the triangular solve involve just one row of the processor template. To avoid these additional sources of overhead, for algorithms such as LU factorization we restrict the class of permissible decompositions further by requiring the blocks to be square. Square block cyclic (SBC) decompositions are parameterized by r , P , and Q , and still make up a large class of widely-used decompositions. For example, column or row decompositions can be achieved by taking P or Q equal to 1, and block decompositions arise if the block size $r = \lceil M/P \rceil$ or $\lceil N/Q \rceil$, where the matrix size is $M \times N$ elements.

Figure 2: Examples of block cyclic decompositions for matrices of size $M \times N$ elements. Each numbered square is a block of $r \times s$ elements, with the number indicating the processor to which that block is assigned. The template is $P \times Q$ processors. The shading is used to emphasize how the template tiles the matrix and has no other significance.

The routines in the LAPACK library are mostly based on the Level 3 BLAS. Thus, we are using distributed memory versions of the Level 3 BLAS as building blocks for the ScaLAPACK library. An important aspect of this approach is that all communication is performed within these distributed Level 3 BLAS so that the upper level source code for a ScaLAPACK routine is identical to that of the corresponding LAPACK routine. This feature enhances portability between distributed memory, shared memory, and vector computers. Users can also use the distributed Level 3 BLAS as a toolkit with which to construct concurrent algorithms not provided in ScaLAPACK. Another advantage of the distributed Level 3 BLAS is that very often they require each processor to sequentially execute a Level 3 BLAS routine. For example, the parallel matrix multiply algorithm consists of a series of steps in each of which each processor multiplies two local matrices by a call to the routine DGEMM. Since highly optimized assembly-coded versions of the sequential Level 3 BLAS already exist on most processors we can take advantage of these in the concurrent versions. Other techniques used to optimize the performance of the distributed Level 3 BLAS include overlapping of communication and calculation where

possible, and the use of assembly-coded routines for packing and unpacking message buffers.

ScaLAPACK will also include a set of Basic Linear Algebra Communication Subprograms (BLACS) for performing different collective communication tasks. These include transforming between different block cyclic decompositions, broadcasting data over rows and columns of the processor template, and performing matrix transposes and circular shifts of arrays.

4 RESULTS

In this section we describe the ScaLAPACK version of the block-partitioned right-looking variant of the LU factorization algorithm, and present performance and scalability results for runs performed on the Intel Touchstone Delta system. The distributed LU factorization algorithm is similar to that implemented on the Intel iPSC/860 and Delta multicomputers by van de Geijn [11]. Given a square matrix, A , of $M_b \times M_b$ blocks, each consisting of $r \times r$ elements, the algorithm generates the factorization $A = LU$ in M_b steps, where U is an upper triangular matrix and L is a lower triangular matrix with 1's on the diagonal. The algorithm is readily extended to the case of nonsquare matrices. After k steps the first kr columns of L and the first kr rows of U have been evaluated, and the matrix A has been updated to the form shown in Figure 3 in which panel B is $(M_b - k) \times 1$ blocks, and C is $1 \times (M_b - k - 1)$ blocks. The next step proceeds as follows,

1. factor B to form the next panel of L , performing partial pivoting over rows if necessary. This evaluates the matrices L_0 , L_1 , and U_0 in Figure 3.
2. solve the triangular system $L_0 U_1 = C$ to get the next row of blocks of U
3. do a rank- r update on the trailing submatrix E , replacing it with $E' = E - L_1 U_1$.

In general, each of these three phases involves inter-processor communication. When factoring B only the P processors in a single column of the processor template are involved in the computation, giving rise to load imbalance. For each of the r columns in turn the pivot is found by first having each processor locate a pivot candidate. We can regard the r elements in the panel row containing the pivot candidate and the index labeling that row as comprising a data structure, $\mathcal{D}$, of $r + 1$ numbers. Each processor's version of $\mathcal{D}$

Figure 3: Schematic diagram showing how the column of blocks B , the row of blocks C , and the trailing submatrix E , are updated in one step of the blocked version of LU factorization.

is input to a logarithmic algorithm that selects the pivot. After this each of the P processors involved in selecting the pivot knows the index of the pivot row, I_{piv} , and the values of the r elements in the panel row containing the pivot (this is the first row of U_0 in Figure 3). The index of the pivot row is pipelined across the processor template, and the other processors perform the pivoting. While this is going on, the P panel processors complete the pivoting within the panel by overwriting row I_{piv} of B by the first row. Since all the panel processors already contain the pivot row, the trailing submatrix part of B can be updated with no further communication. Note how the communication of the pivot location to the other processors, and the pivoting by these processors, is performed while the panel processors are working on the panel factorization. Thus, it is not true to say that the other processors are completely idle during the panel factorization. Finally the panel processors pivot the blocks that they contain lying outside the panel.

After factoring B , the panel is pipelined across the processor template. The Q processors containing the horizontal panel C can then solve the lower triangular system $L_0 U_1 = C$ to find U_1 which is then broadcast down the columns of the template using a spanning tree algorithm. Each processor then performs the rank- r update with no further communication being required.

Initial experiments investigated the optimal block size, r , and showed that over a wide range of problem size and processor template configurations a value of $r = 5$ is close to optimal. This is in agreement with the earlier work of van de Geijn [11]. In all of our subsequent experiments a block size of $r = 5$ was used.

We next consider how performance depends on the configuration of the processor template. For a given number of processors an increase in the number

Figure 4: Performance in gigaflop/s as a function of matrix size for different processor templates containing approximately equal numbers of processors.

of rows, P , in the processor template decreases the amount of computation per processor in the panel factorization, but increases that in the triangular solve phase. Thus, if the communication time were negligible the optimal aspect ratio, P/Q , of the processor template would equal the ratio of the sequential computation times of the panel factorization and triangular solve phases. The actual optimal aspect ratio depends on the communication characteristics of the hardware, and the extent to which communication can be overlapped with computation. We measured the performance for a number of different processor template configurations and problem sizes and found that an aspect ratio, P/Q , of between $1/4$ and $1/8$ to be optimal, and that performance depends rather weakly upon the aspect ratio, particularly at large grain sizes. Some typical results are shown in Figure 4 for 256 processors, which shows a variation of less than 20% in performance as P/Q varies between $1/16$ and 1, for the largest problem. Measured times were converted to gigaflop/s by assuming an operation count of $2M^3/3$, where M is the size of the matrix.

Figure 5 shows the performance as a function of problem size for differing numbers of processors. In all cases the block size is $r = 5$, and we plot results for the processor template that gave the best performance for a given number of processors. The highest performance of 11.8 Gflop/s was attained for a 8×64 processor template and a matrix size of $M = 26000$. This is close to the value of 14 Gflop/s reached by van de Geijn's implementation for a slightly smaller problem ($M = 25000$). We expect to be able to optimize our implementation further.

Figure 5: Performance in gigaflop/s as a function of matrix size for different numbers of processors.

The results in Figure 5 can be used to assess the scalability of our distributed block LU factorization algorithm. As discussed in Section 2, the scalability of an algorithm can be investigated by plotting isoefficiency curves in the (N_p, N) plane. To do this we must determine for a given number of processors the problem size that will result in a specified efficiency. This is rather difficult to do in practice, and usually involves measuring the efficiency for a grid of (N_p, N) configurations and interpolating. A more convenient approach is to look at how the performance per processor degrades as the number of processors increases for a fixed grain size, i.e., by plotting *isogranularity* curves in the (N_p, G) plane, where G is the performance in gigaflop/s. From Equation 2 it is apparent that scalability can readily be assessed by the extent to which the isogranularity curves differ from linearity. The data in Figure 5 can be used to generate the isogranularity curves shown in Figure 6 which show that on the Delta system the scalability starts to degrade for granularity $g < 0.195 \times 10^6$. Since $g = N/N_p = M^2/N_p$, this corresponds to a matrix size of $M = 10000$ on 512 processors.

5 APPLICATIONS

According to a survey into the use of large dense linear system solvers by Edelman [12] these systems mostly arise in the solution of boundary integral equations. Such equations are solved using the boundary element method (BEM). BEM is used, for example, in fluid dynamics computations in the aerospace industry [13], in electromagnetic scattering computations of radar cross-sections [14], in studies of crack

Figure 6: Isogranularity curves in the (G, N_p) plane. The curves are labeled by the granularity g in units of 10^6 .

propagation [15], and in modeling the scattering and transmission of elastic waves in geometries containing multiple cracks [16]. In general, the elements of the coefficient matrix are complex. The LU factorization routine from the ScaLAPACK library has been used by Flanery and Gray to solve a two-dimensional elastodynamics problem using BEM on a 128-processor Intel iPSC/860 hypercube [17]. The goal of this work is to use seismic waves as a diagnostic tool for understanding the process of in situ vitrification, a potentially important technology for remediating hazardous waste sites. The most complex test problem involved the scattering of elastic waves from an array of cylinders, resulting in a relatively small 2048×2048 coefficient matrix. Much larger problems will be considered in the future, with three-dimensional simulations of this type leading to dense matrices of dimensions exceeding 100,000. In the code of Flanery and Gray the assembly of the coefficient matrix is a perfectly parallel operation, while the evaluation of the single righthand side requires some local communication. Performance studies are underway and will be reported in [17].

Concurrent BEM codes use the top level routines of the ScaLAPACK library, but we also expect the distributed Level 3 BLAS to be used as a toolkit to build routines not provided in ScaLAPACK. For example, Sorensen and Vu [18] are using the concurrent version of DGEMM to perform a distributed matrix-matrix multiplication in their largescale sparse eigen-solver code.

6 CONCLUSIONS

In developing the ScaLAPACK library for performing dense and banded matrix computations on distributed memory concurrent computers two key design decisions have been made.

1. Distributed versions of the Level 3 BLAS are used as building blocks, and all interprocessor communication is hidden within these routines. Above the level of the Level 3 BLAS most of the ScaLAPACK code is identical to that of the corresponding LAPACK code for sequential and shared memory machines. By formulating computations in terms of Level 3 BLAS routines the number of messages, and hence the communication latency, is reduced.
2. The block cyclic decomposition scheme is used to distribute the data. This is simple, but sufficiently general-purpose for most applications, and is parameterized by the block size, $r \times s$ elements, and the number of processors, P and Q , in each direction of the processor template. If desired, the user can experiment with these parameters to optimize an application on a particular machine. For certain algorithms, such as in our LU factorization code, we impose the additional requirement that the blocks be square in order to reduce the complexity of the software and communication costs.

We also intend to provide an optional object-based interface to the ScaLAPACK routines will make the library easier to use, and interface naturally with future parallel object-oriented languages. The object-based interface to ScaLAPACK treats matrices as objects. Each matrix is actually a data structure that not only contains a pointer to where the matrix elements are stored, but also contains a complete description of how the matrix is laid out in the hierarchical memory of the concurrent computer. Thus, matrices are passed to ScaLAPACK routines by descriptor, as in CM Fortran, and once the data decomposition has been specified the user does not need to refer to it again when calling ScaLAPACK routines.

A distributed LU factorization algorithm that uses the distributed Level 3 BLAS routines and a square block cyclic decomposition has been implemented on the Intel Delta system. The performance attained is comparable with that obtained with hand-optimized code. Furthermore, our LU factorization algorithm exhibits good scalability on the Delta system if more than about 20% of each processor's memory is utilized.

Future work will focus on completing the implementation of the distributed Level 3 BLAS routines, and developing the object-based interface to the ScaLAPACK routines. We also intend to incorporate the distributed Level 3 BLAS into a number of sparse matrix algorithms currently under development by our collaborators.

Acknowledgements

This research was supported in part by ARPA under contract number DAAL03-91-C-0047 administered by ARO, and in part by DOE under contract number DE-AC05-84OR21400. The research was performed in part using the Intel Touchstone Delta System operated by the California Institute of Technology on behalf of the Concurrent Supercomputing Consortium. Access to this facility was provided through the Center for Research on Parallel Computing.

References

- [1] G. Fox, S. Hiranandani, K. Kennedy, C. Koebel, U. Kremer, C-W. Tseng, and M-Y. Wu, "Fortran D Language Specification," Technical Report CRPC-TR90079, Center for Research on Parallel Computation, Rice University, December 1990.
- [2] A. J. Skjellum and C. Baldwin, "The Multicomputer Toolbox: Scalable Parallel Libraries for Large-Scale Concurrent Applications," Technical Report UCRL-JC-109251, Numerical Mathematics Group, Lawrence Livermore National Laboratory, December 1991.
- [3] J. J. Dongarra, R. van de Geijn, and D. W. Walker, "A Look at Scalable Dense Linear Algebra Libraries," in Proceedings of the 1992 Scalable High Performance Computing Conference, ed. J. Saltz, pub. IEEE Press, 1992.
- [4] Z. Bai and J. Demmel, "Design of a Parallel Nonsymmetric Eigenroutine Toolbox," in preparation.
- [5] J. Demmel, J. J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, and D. Sorensen, "Prospectus for the Development of a Linear Algebra Library for High Performance Computers," Technical Report 97, Argonne National Laboratory, Mathematics and Computer Science Division, September, 1987.

- [6] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, "LAPACK: A Portable Linear Algebra Library for High-Performance Computers," in Proceedings of Supercomputing '90, pp. 1-10, pub. IEEE Press, 1990.
- [7] E. Anderson, Z. Bai, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, *LAPACK Users' Guide*, SIAM Press, 1992.
- [8] J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. van der Vorst, *Solving Linear Systems on Vector and Shared Memory Computers*, SIAM Press, 1990.
- [9] A. Gupta and V. Kumar, "On the scalability of FFT on Parallel Computers," in Proceedings of the Frontiers 90 Conference on Massively Parallel Computation, 1990. Also available as Technical Report TR 90-20, Department of Computer Science, University of Minnesota, Minneapolis, MN 55455.
- [10] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff, "A Set of Level 3 Basic Linear Algebra Subprograms," TOMS, 16:1-17, 1990.
- [11] R. A. van de Geijn, "Massively Parallel LINPACK Benchmark on the Intel Touchstone Delta and iPSC/860 Systems," Technical Report TR-91-28, Department of Computer Science, University of Texas at Austin, 1991.
- [12] A. Edelman, "The First Annual Large Dense Linear System Survey," preprint available by anonymous ftp from math.berkeley.edu.
- [13] J. L. Hess, "Panel Methods in Computational Fluid Dynamics," Annual Rev. Fluid Mechanics, 22:255-274, 1990.
- [14] J. J. H. Wang, *Generalized Moment Methods in Electromagnetics*, John Wiley & Sons, New York, 1991.
- [15] D. O. Potyondy, L. J. Gray, and A. R. Ingraffea, "Simulation of 3D Non-Planar Fatigue Crack Growth in a Turbine Blade Root," presented at the 1992 ASME Pressure Vessels and Piping Summer Conference, New Orleans, Louisiana, June 21-25, 1992.
- [16] Y. C. Angel and J. D. Achenbach, "Harmonic Waves in a Solid Containing a Doubly Periodic Array of Cracks," Wave Motion, 9:337-385, 1987.
- [17] R. E. Flanery, J. B. Drake, and L. J. Gray, "Boundary Elements on Distributed Memory Architectures," preprint, Mathematical Sciences Section, Oak Ridge National Laboratory, P. O. Box 2008, Oak Ridge, TN 37831-6367, U.S.A., 1992.
- [18] D. Sorensen and P. Vu, "Implementation of the Implicitly Restarted Arnoldi Method," preprint, Department of Mathematics, Rice University, 1992.