References

[1] 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*, pages 1–10. IEEE Press, 1990.

[2] 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, Philadelphia, PA, 1992.

[3] E. Anderson, A. Benzoni, J. Dongarra, S. Moulton, S. Ostrouchov, B. Tourancheau, and R. van de Geijn. LAPACK for distributed memory architectures: Progress report. In *Parallel Processing for Scientific Computing, Fifth SIAM Conference*. SIAM, 1991.

[4] C. C. Ashcraft. The distributed solution of linear systems using the torus wrap data mapping. Engineering Computing and Analysis Technical Report ECA-TR-147, Boeing Computer Services, 1990.

[5] R. Brent. The LINPACK benchmark on the AP 1000: Preliminary report. In *Proceedings of the 2nd CAP Workshop*, NOV 1991.

[6] J. Choi, J. J. Dongarra, and D. W. Walker. The design of distributed level 3 BLAS routines, 1992. in preparation.

[7] E. F. Van de Velde. Data redistribution and concurrency. *Parallel Computing*, 16, December 1990.

[8] 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.

[9] J. Dongarra and S. Ostrouchov. LAPACK block factorization algorithms on the Intel iPSC/860. Technical Report CS-90-115, University of Tennessee at Knoxville, Computer Science Department, October 1990.

[10] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, 1990.

[11] J. J. Dongarra, R. van de Geijn, and D. W. Walker. A look at scalable dense linear algebra libraries. In J. H. Saltz, editor, *Proceedings of the 1992 Scalable High Performance Computing Comference*. IEEE Press, 1992.

[12] J.J. Dongarra. Workshop on the BLACS. LAPACK Working Note 34, Technical Report CS-91-134, University of Tennessee, 1991.

[13] J.J. Dongarra and R.A. van de Geijn. Reduction to condensed form for the eigenvalue problem on distributed memory architectures. LAPACK Working Note 30, Technical Report CS-91-130, University of Tennessee, 1991. To appear in *Parallel Computing*.

[14] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, 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.

[15] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker. *Solving Problems on Concurrent Processors*, volume 1. Prentice Hall, Englewood Cliffs, N.J., 1988.

[16] A. Gupta and V. Kumar. On the scalability of FFT on parallel computers. In *Proceedings of the Frontiers 90 Conference on Massively Par allel Computation*, October 1990. Also available as a technical report TR 90-20 from the Computer Science Department, University of Minnesota, Minneapolis, MN 55455.

[17] Y. Saad and M. H. Schultz. Parallel direct methods for solving banded linear systems. Technical Report YALEU/DCS/RR-387, Department of Computer Science, Yale University, 1985.

[18] A. Skjellum and A. Leung. LU factorization of sparse, unsymmetric, Jacobian matrices on multicomputers. In D. W. Walker and Q. F. Stout, editors, *Proceedings of the Fifth Distributed Memory Concurrent Computing Comference*, pages 328–337. IEEE Press, 1990.

[19] R.A. van de Geijn. Massively parallel LINPACK benchmark on the Intel Touchstone Delta and iPSC/860 systems. Computer Science report TR-91-28, Univ. of Texas, 1991.

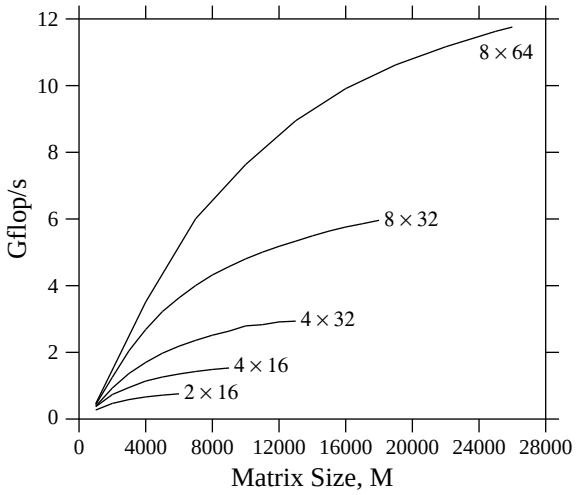

Figure 4: Performance in gigaflop/s as a function of matrix size for different numbers of processors.

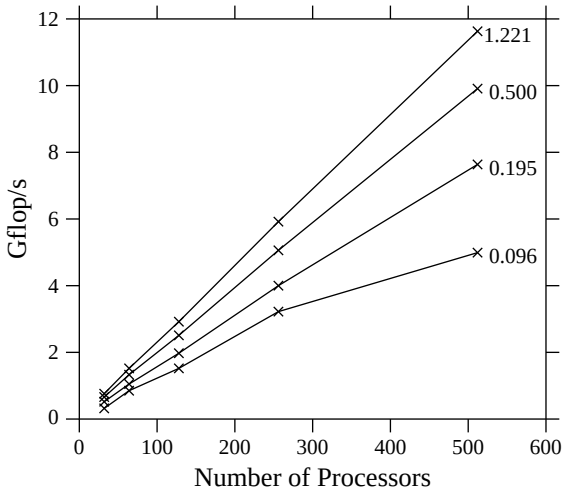

Figure 5: Isogranularity curves in the (G, N_p) plane. The curves are labeled by the granularity g in units of 10^6.

this corresponds to a matrix size of $M = 10000$ on 512 processors.

6 Conclusions

In developing the ScaLAPACK library for performing dense and banded matrix computations on distributed memory concurrent computers three 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 square block scattered decomposition scheme is used to distribute the data. This is simple, but sufficiently general-purpose for most applications. An SBS decomposition is parameterized by the block size, r, 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.

3. An object-based interface to the ScaLAPACK routines will make the library easier to use. 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 an SBS 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 15% of each processor's memory is utilized.

Future work will include further optimization of the distributed LU factorization code. In particular, in the panel factorization it may be possible to increase the overlap of communication with computation by pipelining columns of L across the processor template as soon as they are evaluated, rather than pipelining all of the panel across after factoring it. We shall also focus on completing the implementation of the distributed Level 3 BLAS routines, and developing the object-based interface to the ScaLAPACK routines.

Acknowledgments

This 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.

5 Results for the Distributed LU Factorization Algorithm

In this section we present performance results for an implementation of the distributed LU factorization algorithm described above on the Intel Touchstone Delta system. The Delta system is a distributed memory MIMD computer containing 520 i860-based compute nodes connected via a two-dimensional communication network. 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 [19]. 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 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 Fig. 3 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 4 shows the performance as a function of problem size for differing numbers of nodes. 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.

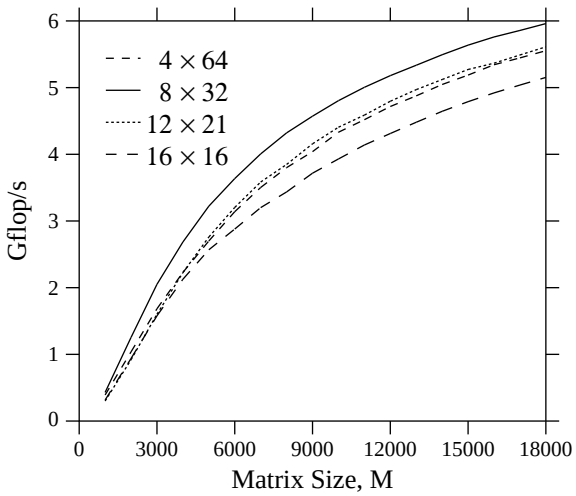

Figure 3: Performance in gigaflop/s as a function of matrix size for different processor templates containing approximately equal numbers of processors.

The results in Fig. 4 can be used to assess the scalability of our distributed block LU factorization algorithm. In general one would expect the concurrent efficiency of a given algorithm on a given machine to depend on the problem size, N, and the number of processors used, N_p. Thus,

$$\epsilon(N, N_p) = \frac{1}{N_p} \frac{T_1(N)}{T(N, N_p)} \tag{3}$$

where $T(x, y)$ is the time for a problem of size x to run on y processors, $T_1(x)$ is the time to run on one processor using the best sequential algorithm. An algorithm is perfectly scalable if the concurrent efficiency depends only on the grain size, g, and not independently on N and N_p. The efficiency could be investigated by plotting isoefficiency curves in the (N_p, N) plane [16]. For a highly scalable algorithm these curves would be straight lines. A more useful 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. Since

$$G \propto \frac{T_1(N)}{T(N, N_p)} = N_p \, \epsilon(N, N_p) \tag{4}$$

scalability can readily be assessed by the extent to which the isogranularity curves differ from linearity. The data in Fig. 4 can be used to generate the isogranularity curves shown in Fig. 5 which show that on the Delta system the scalability starts to degrade for granularity $g < 0.195 \times 10^6$. Since $g = N/N_p = M^2/N_p$,

BLAS, (2) the LACS, and (3) assembly coded routines for performing common sequential Level 3 BLAS operations, and tasks such as buffer copying. All communication in a ScaLAPACK library routine is performed within the distributed Level 3 BLAS, and so the user is isolated from the details of the parallel implementation. An important consequence of this is that the source code of the higher level routines, for example for the LU, QR, and Cholesky factorizations, is identical in the ScaLAPACK and LAPACK libraries. LU factorization involves calls to the Level 3 BLAS routines DGEMM, for performing matrix multiplication, and DTRSM, for solving triangular systems. Details of the distributed implementation of these Level 3 BLAS routines is given in [6]. In the LAPACK version of the right-looking LU factorization algorithm DGEMM and DTRSM are used to apply a rank-r update to the trailing submatrix, and to perform the lower triangular solve necessary to form the block rows of U, the upper triangular matrix obtained by the factorization. We have implemented a distributed LU factorization routine that makes use of the SBS data distribution and the distributed Level 3 BLAS routines, and are in the process of incorporating a object-based interface. 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 Fig. 2 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 Fig. 2.

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 interprocessor 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 $\mathcal{D}$ 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 Fig. 2). 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.

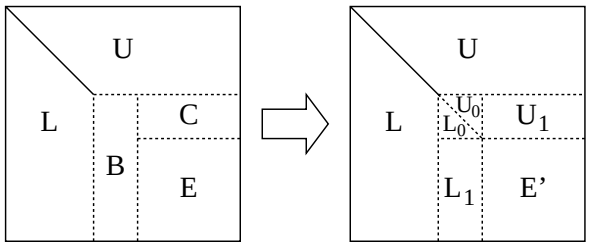

Figure 2: Schematic diagram showing how the column of blocks B, the row of blocks C, and the trailing submatrix E, are updated in one step of the blocked version of LU factorization.

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.

```
MATRIX {
    MATRIX_DATA_PART_PTR {
        type    *matrix_elements;    % pointer to the matrix element values
        int      MG_elements;        % total number of rows in matrix
        int      NG_elements;        % total number of columns in matrix
        int      MG_blocks;          % total number of rows of r by r blocks in matrix
        int      NG_blocks;          % total number of columns of r by r blocks in matrix
        int      M_blocks;           % number of rows of r by r blocks in each processor
        int      N_blocks;           % number of columns of r by r blocks in each processor
        int      row_temp_offset;    % the template row containing the first matrix block
        int      col_temp_offset;    % the template column containing the first matrix block
        char    *user_data;          % pointer to user-supplied buffer
    }
    DECOMPOSITION_PART_PTR {
        int      r;                  % the block size
        int      P;                  % the number of rows of processors in the template
        int      Q;                  % the number of columns of processors in the template
        int      left_proc;          % the ID number of the processor to the left in the template
        int      right_proc;         % the ID number of the processor to the right in the template
        int      below_proc;         % the ID number of the processor below in the template
        int      above_proc:         % the ID number of the processor above in the template
    }
    STORAGE_PART_PTR {
        int      elements_by_col;    % indicates if blocks are stored by column or row
        int      blocks_by_col;      % indicates if elements in block are stored by column or row
        int      row_e_offset;       % offset between successive elements in same row
        int      col_e_offset;       % offset between successive elements in same column
        int      row_b_offset;       % offset between start of successive blocks in same row
        int      col_b_offset;       % offset between start of successive blocks in same column
    }
}
```

Figure 1: The matrix object data structure in a C-like pseudocode. The matrix object consists of three pointers; one to each of the matrix data, decomposition, and storage parts, the contents of which are as shown. The data type of the matrix elements may be real, complex, double precision real, or double precision complex.

align matrix B with the decomposition, as before. The second method is used to create a submatrix of an existing matrix. In this case, we specify the start and extent of the submatrix, which then inherits the decomposition and storage parts from its parent matrix. In the discussion so far it has been assumed that the matrix elements have been assigned values, either in some previous computational phase, or by reading in values from disk. As a convenience, we supply a third matrix creation method that generates a random matrix. This is done by a subroutine that takes as input a random number seed, the size of the matrix, and pointers to previously created decomposition and storage part data structures. The subroutine allocates storage for the random matrix, and returns the matrix object. A leap-frog method [15] is used to generate the random numbers, so for a given matrix size and seed the matrix is the same for all processor templates.

The subroutines that specify the decomposition and storage only assign values in these data structures. They do not change the decomposition. We call these *assignment* subroutines. Another set of *inquiry* subroutines may be used to extract information from the matrix data structure. These inquiry routines might be used by an application programmer wishing to perform some task for which there is no appropriate library routine, or by someone wanting to extend ScaLAPACK. A set of Linear Algebra Communication Subroutines (LACS) may be used to transform the decomposition of a matrix [12].

4 Parallel LU Factorization

The ScaLAPACK library is built using three types of routine; (1) distributed versions of the Level 3

This results in scattered blocks of size $r \times s$. We can view the block scattered decomposition as stamping a $P \times Q$ processor grid, or template, over the matrix, where each cell of the grid covers $r \times s$ data items, and is labeled by its position in the template. The block and scattered decompositions may be regarded as special cases of the block scattered decomposition. In general, the scattered blocks are rectangular, however, the use of nonsquare blocks can lead to complications, and additional concurrent overhead. We, therefore, propose to restrict ourselves to the square block scattered (SBS) class of decompositions. The column and row decompositions can still be recovered by setting $P = 1$ or $Q = 1$. However, more general decompositions for which $r \neq s$, and neither P nor Q is 1, cannot be reproduced by a SBS decomposition. These types of decomposition are not often used in matrix computations.

The SBS decomposition scheme is practical and sufficiently general-purpose for most, if not all, dense linear algebra computations. Furthermore, in problems, such as LU factorization, in which rows and/or columns are eliminated in successive steps, the SBS decomposition enhances scalability by ensuring statistical load balance.

So far we have only considered how to map matrix elements onto the processor template. In decomposing a problem we must also specify how locations in the processor template are mapped to physical processors. On most current multicomputers the cost of communicating between any two processors is weakly dependent on their separation in the topology of the communication network. Hence the choice of mapping should not impact performance very much. ScaLAPACK supports the natural and Gray code mappings, as well as any mapping function supplied by the application programmer.

3 An Object-Oriented Library Interface

In ScaLAPACK matrices are objects. In other words, a matrix is a data structure containing information that completely describes the matrix, and its decomposition. Thus, when calling a ScaLAPACK routine only the names of the matrices involved need be supplied in the subroutine argument list. Details such as the matrix size and decomposition need not be given explicitly as subroutine arguments.

The matrix object consists of three parts; a *matrix data* part, a *decomposition part*, and a *storage part*. The matrix data part is a pointer to a data structure that contains a pointer to the start of the matrix element values for a processor, together with data about the size of the matrix, and the position in the processor mesh of the first block in the distributed matrix. Another pointer specifies work space supplied by the application programmer for storing data generated during a library call, such as the pivot sequence generated by partial pivoting in LU factorization. The decomposition part is a pointer to a data structure giving the square block size, r, and the number of rows, P, and columns, Q, in the processor mesh (or *template*). In addition, the decomposition part data structure contains the ID numbers of the four neighboring processors in the processor mesh. The storage part is a pointer to a structure that specifies how the matrix data are stored in each processor. For example, whether the data in each block are stored by columns or rows, and whether the blocks in each processor are stored by columns or rows. The memory offsets in matrix elements between an element and the next element in the same row, and the next element in the same column are also contained in this data structure, together with the memory offsets between corresponding elements in adjacent blocks. The specifications of the matrix object are given in Fig. 1. As our research progresses we expect to make further changes in the content of the matrix object data structure. We intend to use Fortran 90 to implement this object-based interface, and are investigating the use of preprocessors that will allow us to develop a truly object-oriented library interface.

To create a matrix the decomposition must first be specified. A routine is called that returns a pointer to a `DECOMPOSITION_PART` data structure. The values of the block size, the size of the processor template, and the ID numbers of the neighboring processors in the template are then filled in by a series of subroutine calls. A similar procedure is followed to create the `STORAGE_PART` data structure. Once the decomposition part has been created a subroutine is called to align the matrix with the decomposition. This involves specifying the location in the template that contains the first block in the matrix. Another subroutine call is used to associate the previously created `STORAGE_PART` data structure with the matrix. A final subroutine call instantiates the matrix, and fills in the rest of the `MATRIX_DATA_PART` data structure.

There are currently three other ways of creating matrices. In the first method, we specify that some matrix, B, has the same decomposition and storage parts as some previously created matrix, A, and then

be extended to the Paragon and CM-5 once we have access to stable systems.

2 Square Block Scattered Data Decomposition

The layout of an application's data within the hierarchical memory of a concurrent computer is critical in determining the performance and scalability of the parallel code. On shared memory concurrent computers (or *multiprocessors*) the software package LAPACK [1,8] 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.

On a multicomputer the application programmer is responsible for decomposing the data over the processors of the concurrent computer. A vector of length M may be decomposed over some set of N_p processors by first arranging the processors in a linear sequence, and then assigning the vector entry with global index m (where $0 \leq m < M$) to the pth processor in the sequence ($0 \leq p < N_p$), where it is stored as the ith entry in a local array. Thus the decomposition of a vector can be regarded as a mapping of the global index, m, to an index pair, (p, i), specifying the processor location and the local index.

For matrix problems one can think of arranging the processors as a P by Q grid. Thus the grid consists of P rows of processors and Q columns of processors, and $N_p = PQ$. Each processor can be uniquely identified by its position, (p, q), on the processor grid. The decomposition of an $M \times N$ matrix can be regarded as the tensor product of two vector decompositions, μ and ν. The mapping μ decomposes the M rows of the matrix over the P rows of processors, and ν decomposes the N columns of the matrix over the Q columns of processors. Thus, if $\mu(m) = (p, i)$ and $\nu(n) = (q, j)$ then the matrix entry with global index (m, n) is assigned to the processor at position (p, q) on the processor grid, where it is stored in a local array with index (i, j).

Two common decompositions are the *block* and the *scattered* decompositions [7,15]. The block decomposition, λ, assigns contiguous entries in the global vector to the processors in blocks.

$$\lambda(m) = (\lfloor m/L \rfloor , m \bmod L) , \tag{1}$$

where $L = \lceil M/P \rceil$. The scattered decomposition, σ, assigns consecutive entries in the global vector to different processors,

$$\sigma(m) = (m \bmod P, \lfloor m/P \rfloor) \tag{2}$$

By applying the block and scattered decompositions over rows and columns a variety of matrix decompositions can be generated.

The *block scattered* decomposition scatters blocks of r elements over the processors instead of single elements, and if the blocks are rectangular, is able to reproduce the decompositions resulting from all possible block and scattered decompositions. Thus, by using the block scattered decomposition a large degree of decomposition independence can be attained. In the block scattered decomposition the mapping of the global index, m, can be expressed as a triplet of values, $\mu(m) = (p, t, i)$, where p is the processor position, t the block number, and i the local index within the block. For the block scattered decomposition we may write,

$$\zeta_r(m) = \left(\left\lfloor \frac{m \bmod T}{r} \right\rfloor , \left\lfloor \frac{m}{T} \right\rfloor , \Big(m \bmod T \Big) \bmod r \right) \tag{3}$$

where $T = rP$. It should be noted that this reverts to the scattered decomposition when $r = 1$, with local block index $i = 0$. A block decomposition is recovered when $r = L$, with block number $t = 0$. The block scattered decomposition in one form or another has previously been used by Saad and Schultz [17], Skjellum and Leung [18], Dongarra and Ostrouchov [9], Anderson et al. [3], Ashcraft [4], Dongarra and van de Geijn [13], van de Geijn [19], and Brent [5], to name a few. The block scattered decomposition is one of the decompositions provided in the Fortran D programming style [14].

As discussed above, the block scattered decomposition of a matrix can be regarded as the tensor product of two block scattered decompositions, μ_r and ν_s.

ScaLAPACK: A Scalable Linear Algebra Library for Distributed Memory Concurrent Computers*

Jaeyoung Choi[§], Jack J. Dongarra[§‡], Roldan Pozo[‡], and David W. Walker[§]

[§]Oak Ridge National Laboratory
Mathematical Sciences Section
P. O. Box 2008, Bldg. 6012
Oak Ridge, TN 37831-6367

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

Abstract

This paper describes ScaLAPACK, a distributed memory version of the LAPACK software package for dense and banded matrix computations. Key design features are the use of distributed versions of the Level 3 BLAS as building blocks, and an object-based interface to the library routines. The square block scattered decomposition is described. The implementation of a distributed memory version of the right-looking LU factorization algorithm on the Intel Delta multicomputer is discussed, and performance results are presented that demonstrate the scalability of the algorithm.

1 Introduction

This paper considers issues in the design and implementation of a library of subroutines for performing linear algebra computations on distributed memory concurrent computers (or *multicomputers*). When completed the library will contain subroutines for performing dense, banded, and sparse matrix computations, with the latter being divided into the symmetric, positive-definite, and nonsymmetric cases. In this paper we focus on ScaLAPACK, a distributed memory version of the LAPACK [1,2] software package for dense and banded matrix problems. Among the important design goals are scalability, portability, flexibility, and ease-of-use. In the context of the current work an algorithm is regarded as "scalable" if it continues to perform efficiently the task for which it was designed as the number of processors increases, while keeping the granularity fixed. The intent is that for large-scale problems the library routines should effectively exploit the computational hardware of medium grain-size multicomputers with up to a few thousand processors, such as the Intel Paragon and Thinking Machines Corporation's CM-5.

Scalability is largely determined by how the algorithm interacts with the multicomputer hardware and low-level software, and so reduces to an algorithm design issue, from our point of view. Issues such as load balance, communication volume, and whether communication and computation can be overlapped, all impact the scalability of an algorithm and must be carefully considered. The way in which the data are distributed (or decomposed) over the processors of the multicomputer is of fundamental importance to these factors.

We shall use the term "programmability" to refer to factors such as portability, flexibility, and ease-of-use. Programmability is largely determined by how the user interacts with the software library. To enhance the programmability of the library we would like details of the parallel implementation to be hidden as much as possible from the user, and so have designed an object-based interface to the library. This is described in Sec. 3. In addition, it is desirable for the software to work correctly for a large class of data decompositions. We have, therefore, adopted the square block scattered (SBS) decomposition, described in more detail in Sec. 2, for use in all our distributed dense linear algebra algorithms. In Sec. 4, we describe a distributed right-looking variant of the LU factorization algorithm. The scalability of the algorithm is demonstrated in Sec. 5 by experiments on the Intel Delta multicomputer. These experiments will

*This work was supported in part by DARPA and ARO under contract number DAAL03-91-C-0047, and in part by DOE under contract number DE-AC-05-84OR21400