LAPACK Working Note 43
A Look at Scalable Dense Linear Algebra Libraries *

Jack Dongarra †‡
Robert van de Geijn §
David Walker ‡

†Dept. of Computer Science
University of Tennessee
Knoxville, TN 37996

‡Mathematical Sciences Section
Oak Ridge National Laboratory
Oak Ridge, TN 37831

§Department of Computer Sciences
University of Texas
Austin, TX 78712

May 5, 1992

Abstract

We discuss the essential design features of a library of scalable software for performing dense linear algebra computations on distributed memory concurrent computers. The square block scattered decomposition is proposed as a flexible and general-purpose way of decomposing most, if not all, dense matrix problems. An object-oriented interface to the library permits more portable applications to be written, and is easy to learn and use, since details of the parallel implementation are hidden from the user. Experiments on the Intel Touchstone Delta system with a prototype code that uses the square block scattered decomposition to perform LU factorization are presented and analyzed. It was found that the code was both scalable and efficient, performing at about 14 GFLOPS (double precision) for the largest problem considered.

*This work was supported by DARPA and ARO under contract number DAAL03-91-C-0047, by the National Science Foundation Science and Technology Center Cooperative Agreement No. CCR-8809615, and by Intel Supercomputing Systems Division. This paper is a corrected version of a paper that appeared in the Proceedings of the Scalable High Performance Computing Conference (SHPCC-92), Williamsburg, VA, April 26-29, 1992. To receive a postscript copy, send email to `netlib@ornl.gov` and in the mail message type: `send lawn43.ps from lapack/lawns`.

1 Introduction

Advanced parallelizing compilers may one day be capable of generating efficient parallel code for MIMD distributed memory concurrent computers (or *multicomputers*) from sequential code. However, in the interim, the development of scalable libraries is a key component in the development of a software environment that will allow the computational power of multicomputers to be exploited, and made available to a broader community of users. Over the next few years we envisage such libraries being developed in a number of areas, and that they will be accessible through a variety of interfaces. This paper focuses on issues impacting the design of scalable libraries for performing dense linear algebra on multicomputers. However, we believe that many of the issues discussed here are applicable to scalable libraries in other areas, and, indeed, it is important to impose some uniformity upon the design of different libraries.

In the next section we discuss data allocation, that is, how the data items in a parallel program are laid out in the hierarchical memory of the concurrent computer. The block scattered decomposition will be shown to encompass a large class of decompositions, and to provide sufficient flexibility for essentially all dense linear algebra computations. In Section 3 we use the right-looking variant of the LU factorization algorithm for dense matrices to demonstrate the block scattered decomposition for a specific well-known example. A brief discussion of the run-time analysis of the algorithm is given, together with results of experiments running at up to 14 GFLOPS on the Intel Touchstone Delta system. Section 4 deals with programmability and implementation issues, and will discuss an objected-oriented approach to scalable libraries. Conclusions are presented in Section 5.

2 Data Allocation

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*) there are at least three levels to the memory hierarchy: the shared memory, and each processor's cache and registers. On such machines efficient codes seek to maximize the cache hit ratio, i.e., to avoid having to reload the cache too frequently. The software package LAPACK [1, 8] does this by casting linear algebra computations in terms of block-oriented, matrix-matrix operations known as the Level 3 BLAS [10, 11] whenever possible. This approach generally results in high cache hit ratios, without requiring any explicit cache manipulation by the application programmer. One of the aims of our work is to investigate a distributed memory version of LAPACK.

There are also levels to the memory hierarchy on multicomputers: the local and nonlocal (remote) memory. In addition, each processor may have a hierarchical memory. Each processor has its own local memory, and the nonlocal memory for a given processor is simply the local memory of the other processors. A processor plus its local memory and other closely coupled hardware is refered to as a *node*. The nodes of a multicomputer are connected via a communication network; there is no physical shared memory. There are two important differences between multiprocessors and multicomputers. The first is that multiprocessors are generally faster than multicomputers in transfering data between two layers of the memory hierarchy. In particular, MIMD multicomputers typically incur a high communication latency. The second difference is that while bus-based multiprocessors usually have no more than 20 or 30 processors, multicomputers typically have several hundred to a few thousand processors. Thus the processors of a multiprocessor are large grain size and closely coupled, whereas those of a multicomputer are of smaller grain size and are less closely coupled. This means that the programming techniques and algorithms that are successful on multiprocessors may not result in scalable codes on multicomputers.

On a multicomputer the application programmer is responsible for distributing (or *decomposing*) the data over the nodes of the concurrent computer. A vector of length M may be decomposed over some set of N_p nodes by first arranging the nodes in a linear sequence, and then assigning the vector entry with global index m (where $0 \le m < M$) to the pth node in the sequence ($0 \le p < N_p$), 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 node location and the local index.

For matrix problems one can think of arranging the nodes as a P by Q grid. Thus the grid consists of P rows of nodes and Q columns of nodes, and $N_p = PQ$. Each node can be uniquely identified by its position, (p,q), on the node 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 nodes, and ν decomposes the N columns of the matrix over the Q columns of nodes. Thus, if $\mu(m) = (p,i)$ and $\nu(n) = (q,j)$ then the matrix entry with global index (m,n) is assigned to the node at position (p,q) on the node grid, where it is stored in a local array with index (i,j).

Two common decompositions are the *block* and the *scattered* decompositions [7, 18]. The block decomposition, λ, assigns contiguous entries in the global vector to the nodes in blocks.

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

where $L = \lfloor (M-1)/P \rfloor + 1$. The scattered decomposition, σ, assigns consecutive entries in the global vector to different nodes,

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

Figure 1 shows examples of these two types of decomposition for a 10×10 matrix.

Two features that are desirable in a parallel subroutine library are;

1. a large degree of decomposition independence, so that a subroutine will work correctly for a large class of decompositions of the input data,
2. a set of communication routines for transforming between different decompositions.

These components give the application programmer the option of changing the decomposition, if necessary, so that a given phase of the computation can be performed optimally, i.e., with the least concurrent overhead. Alternatively, the programmer may choose to leave the decomposition unchanged and perform the computation suboptimally, thereby avoiding the overhead associated with changing the decomposition. The important point here is that the software should be sufficiently flexible to permit the programmer to make the choice, rather than imposing a particular method.

Decomposition-independence could be achieved by having the subroutine contain a conditional statement, with each clause corresponding to a different type of decomposition. A more elegant and, we believe, better approach is to use a block scattered decomposition that is able to reproduce all the decompositions in Fig. 1, except for those shown in Figs. 1(f) and (g). In the block scattered approach blocks of r elements are scattered over the nodes instead of single elements. The mapping of the global index, m, can be expressed as a triplet of values, $\mu(m) = (p,t,i)$, where p is the node 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 [20], Skjellum and Leung

0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0
0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0
0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0
1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0
1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0
1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0
2,0	2,0	2,0	2,0	2,0	2,0	2,0	2,0	2,0	2,0
2,0	2,0	2,0	2,0	2,0	2,0	2,0	2,0	2,0	2,0
2,0	2,0	2,0	2,0	2,0	2,0	2,0	2,0	2,0	2,0
3,0	3,0	3,0	3,0	3,0	3,0	3,0	3,0	3,0	3,0

(a) μ block, P=4, Q=1

0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0
1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0
2,0	2,0	2,0	2,0	2,0	2,0	2,0	2,0	2,0	2,0
3,0	3,0	3,0	3,0	3,0	3,0	3,0	3,0	3,0	3,0
0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0
1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0
2,0	2,0	2,0	2,0	2,0	2,0	2,0	2,0	2,0	2,0
3,0	3,0	3,0	3,0	3,0	3,0	3,0	3,0	3,0	3,0
0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0
1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0

(b) μ scattered, P=4, Q=1

0,0	0,0	0,0	0,1	0,1	0,1	0,2	0,2	0,2	0,3
0,0	0,0	0,0	0,1	0,1	0,1	0,2	0,2	0,2	0,3
0,0	0,0	0,0	0,1	0,1	0,1	0,2	0,2	0,2	0,3
0,0	0,0	0,0	0,1	0,1	0,1	0,2	0,2	0,2	0,3
0,0	0,0	0,0	0,1	0,1	0,1	0,2	0,2	0,2	0,3
0,0	0,0	0,0	0,1	0,1	0,1	0,2	0,2	0,2	0,3
0,0	0,0	0,0	0,1	0,1	0,1	0,2	0,2	0,2	0,3
0,0	0,0	0,0	0,1	0,1	0,1	0,2	0,2	0,2	0,3
0,0	0,0	0,0	0,1	0,1	0,1	0,2	0,2	0,2	0,3
0,0	0,0	0,0	0,1	0,1	0,1	0,2	0,2	0,2	0,3

(c) ν block, P=1, Q=4

0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1
0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1
0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1
0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1
0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1
0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1
0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1
0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1
0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1
0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1

(d) ν scattered, P=1, Q=4

0,0	0,0	0,0	0,1	0,1	0,1	0,2	0,2	0,2	0,3
0,0	0,0	0,0	0,1	0,1	0,1	0,2	0,2	0,2	0,3
0,0	0,0	0,0	0,1	0,1	0,1	0,2	0,2	0,2	0,3
1,0	1,0	1,0	1,1	1,1	1,1	1,2	1,2	1,2	1,3
1,0	1,0	1,0	1,1	1,1	1,1	1,2	1,2	1,2	1,3
1,0	1,0	1,0	1,1	1,1	1,1	1,2	1,2	1,2	1,3
2,0	2,0	2,0	2,1	2,1	2,1	2,2	2,2	2,2	2,3
2,0	2,0	2,0	2,1	2,1	2,1	2,2	2,2	2,2	2,3
2,0	2,0	2,0	2,1	2,1	2,1	2,2	2,2	2,2	2,3
3,0	3,0	3,0	3,1	3,1	3,1	3,2	3,2	3,2	3,3

(e) μ block, ν block, P=Q=4

0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1
0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1
0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1
1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3	1,0	1,1
1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3	1,0	1,1
1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3	1,0	1,1
2,0	2,1	2,2	2,3	2,0	2,1	2,2	2,3	2,0	2,1
2,0	2,1	2,2	2,3	2,0	2,1	2,2	2,3	2,0	2,1
2,0	2,1	2,2	2,3	2,0	2,1	2,2	2,3	2,0	2,1
3,0	3,1	3,2	3,3	3,0	3,1	3,2	3,3	3,0	3,1

(f) μ block, ν scattered, P=Q=4

0,0	0,0	0,0	0,1	0,1	0,1	0,2	0,2	0,2	0,3
1,0	1,0	1,0	1,1	1,1	1,1	1,2	1,2	1,2	1,3
2,0	2,0	2,0	2,1	2,1	2,1	2,2	2,2	2,2	2,3
3,0	3,0	3,0	3,1	3,1	3,1	3,2	3,2	3,2	3,3
0,0	0,0	0,0	0,1	0,1	0,1	0,2	0,2	0,2	0,3
1,0	1,0	1,0	1,1	1,1	1,1	1,2	1,2	1,2	1,3
2,0	2,0	2,0	2,1	2,1	2,1	2,2	2,2	2,2	2,3
3,0	3,0	3,0	3,1	3,1	3,1	3,2	3,2	3,2	3,3
0,0	0,0	0,0	0,1	0,1	0,1	0,2	0,2	0,2	0,3
1,0	1,0	1,0	1,1	1,1	1,1	1,2	1,2	1,2	1,3

(g) μ scattered, ν block, P=Q=4

0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1
1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3	1,0	1,1
2,0	2,1	2,2	2,3	2,0	2,1	2,2	2,3	2,0	2,1
3,0	3,1	3,2	3,3	3,0	3,1	3,2	3,3	3,0	3,1
0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1
1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3	1,0	1,1
2,0	2,1	2,2	2,3	2,0	2,1	2,2	2,3	2,0	2,1
3,0	3,1	3,2	3,3	3,0	3,1	3,2	3,3	3,0	3,1
0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1
1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3	1,0	1,1

(h) μ scattered, ν scattered, P=Q=4

Figure 1: These 8 figures show different ways of decomposing a 10×10 matrix. Each cell represents a matrix entry, and is labeled by the position, (p,q), in the node grid of the node to which it is assigned. To emphasize the pattern of decomposition the matrix entries assigned to the node in the first row and column of the node grid are shown shaded. Figures (a) and (b) show block and scattered row-oriented decompositions, respectively, for 4 nodes arranged as a 4×1 grid ($P = 4$, $Q = 1$). In figures (c) and (d) the corresponding column-oriented decompositions are shown ($P = 1$, $Q = 4$). Figures (e)-(h) show block and scattered decompositions for 16 nodes arranged as a 4×4 grid ($P = Q = 4$).

Figure	P	Q	r	s	$r = s$
(a)	4	1	3	10	3
(b)	4	1	1	10	1
(c)	1	4	10	3	3
(d)	1	4	10	1	1
(e)	4	4	3	3	3
(f)	4	4	3	1	–
(g)	4	4	1	3	–
(h)	4	4	1	1	1

Table 1: Block-scattered decomposition parameters needed to reproduce the block and scattered decompositions in Fig. 1. The last column gives the block size when only square blocks are used. Decompositions (f) and (g) cannot be generated with square blocks.

[21], Dongarra and Ostrouchov [9], Anderson et al. [2], Ashcraft [4, 5], Dongarra and van de Geijn [15], van de Geijn [22], and Brent [6], to name a few. The block scattered decomposition is one of the decompositions provided in the Fortran D programming style [17].

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. 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. In Table 1 we give the values of the block size $r \times s$ that give the same results as the block and scattered decompositions in Fig. 1. 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. For example, in the LU factorization algorithm, described in the next section, a triangular solve is needed to update submatrix C. If nonsquare blocks are used either the triangular matrix will extend over more than one column of blocks (if $r > s$), or the submatrix C will extend over more than one row of blocks (if $r < s$). Thus, nonsquare blocks will result in additional software and communication 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 $Q = 1$, as shown in Table 1, however, the decompositions shown in Figs. 1(f) and (g) cannot be generated with an SBS decomposition.

So far we have only considered how to map matrix elements onto the node grid. In decomposing a problem we must also specify how locations in the node grid are mapped to physical nodes. Common mapping functions are the natural mapping,

$$A(i, j) = i + j \cdot Q \qquad (4)$$

and the binary-reflected Gray code mapping,

$$A(i, j) = G(i) + G(j) \cdot Q \qquad (5)$$

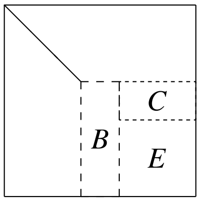
where $G(x)$ denotes the Gray code of x, and $i = 0, 1, \ldots, Q-1$, $j = 0, 1, \ldots, P-1$. On most current multicomputers the cost of communicating between any two nodes is weakly dependent of their separation in the topology of the communication network. Hence the choice of mapping should not impact performance very much. The subroutine library should support the natural and Gray code mappings, as well as any function, A, supplied by the application programmer.

3 An Example

In this section, we discuss the scalability of the LU factorization algorithm when it is implemented using the block scattered decomposition. First, we describe the algorithm. Next, we summarize the results from an analysis of the time complexity. Data from experiments on the Intel Touchstone Delta system are used to further demonstrate the scalability.

3.1 LU factorization

To obtain our parallel implementation of the LU factorization, we started with a variant of the right-looking LAPACK LU factorization routine. It can be briefly described as follows: Assume the LU factorization has proceeded so that all but the labeled portions of the matrix have been updated:



where $B \in \mathbf{R}^{M\times r}$, $C \in \mathbf{R}^{r\times(M-r)}$, and $E \in \mathbf{R}^{(M-r)\times(M-r)}$. During the next step, the right-looking algorithm factors panel B, pivoting if necessary. Next, the pivots are applied to the remainder of the matrix. Blocks C and E now become blocks $\bar{C}$ and $\bar{E}$, a triangular solve updates submatrix $\bar{C}$, and a rank n_b update updates submatrix $\bar{E}$. This process continues recursively with the updated matrix [12].

Turning now to the distributed memory implementation, assume the matrix is distributed among a $P \times Q$ grid on nodes using a block scattered decomposition, with block size $r \times r$. For our analysis, we assume that communicating a block of k floating point numbers between any two nodes requires time $\alpha + k\beta$, where α and β represent the communication latency and the inverse of the bandwidth, respectively. In addition, the time for a floating point operation is given by γ. Finally, in our formulas, $\lceil k \rceil$ indicates the smallest integer *multiple of r* greater than k.

The above described process proceeds as follows:

- (fB) The column of nodes that holds B collaborates to factor this panel. Since there is relatively little to compute (the panel is typically narrow), and communication is restricted to short messages, the contribution of this operation to the run-time is almost entirely due to communication latency. We will ignore the other costs. For each column, this consists of $\log(P)\alpha$ for determining the pivot row, α for swapping pivot rows of this panel, and another $\log(P)\alpha$ for broadcasting the pivot row. (Possible optimization: since this is latency bound, a clever implementation would combine the messages for determining the pivot row, and distributing it within the column of nodes that hold the panel.)

- (bp) Pivot information is distributed to all other columns of nodes. Approximate contribution to run-time: α per panel.

- (p) Columns of nodes collaborate to pivot the remainders of the matrix rows. Approximate contribution to run-time: $r(\alpha + \lceil (N-r)/Q \rceil \beta)$ for each of the N/r panels.

- (b$\bar{B}$) Factored panel B is distributed within rows of nodes. Approximate contribution to run-time: $2(\alpha + \lceil (N-(k-1)r)/P \rceil r\beta)$ for panel $k = 1, \ldots, N/r$. (Since this operation can be pipelined around the ring, overlapping with computation, there is no $\log(Q)$ term here.)

- ($\mathrm{b}\bar{C}$) The row that holds $\bar{C}$ performs the triangular solve, the results of which are distributed within columns of nodes. Approximate contribution: $\lceil (N-kr)/Q \rceil r^2\gamma + \log(P)(\alpha + \lceil N-kr)/Q \rceil r\beta)$ for panel $k = 1, \ldots, N/r$.

- ($\mathrm{u}\bar{E}$) Most parallelism is derived from updating $\bar{E}$. Approximate contribution:

$$2\lceil (N-kr)/P \rceil \lceil (N-kr)/Q \rceil r\gamma$$

  for panel $k = 1, \ldots, N/r$.

The total run time is then given by

$$T_{tot} \approx T_{\mathrm{fB}} + T_{\mathrm{fB}} + T_{\mathrm{bp}} + T_{\mathrm{p}} + T_{\mathrm{b}\bar{\mathrm{B}}} + T_{\mathrm{f}\bar{\mathrm{C}}} + T_{\mathrm{u}\bar{\mathrm{E}}} \tag{1}$$

where the different terms come from summing over all panels the different contributions given above.

Since the total computation time of the algorithm on a single processor is given by $T_1 \approx (2/3)N^3\gamma$, the efficiency attained, $E = T_1/pT_{tot}$, as a function of the various parameters, can be shown to be of the form

$$\begin{aligned} E &= \left[1 + \frac{p}{N^2}\left(c_1 \log(P) + c_2\right)\frac{\alpha}{\gamma}\right. && (2) \\ &\quad \left. + \frac{P}{N}\left(c_3 \log(P)\frac{\beta}{\gamma} + c_4\right) + \frac{Q}{N}\left(c_5\frac{\beta}{\gamma} + c_6\right)\right]^{-1} && (3) \end{aligned}$$

where c_{1-6} depend only on r.

Let us start by considering the block column scattered decomposition, i.e., $P \times Q = 1 \times p$. Then, for reasonably large N,

$$E \approx \left[1 + c_2\frac{p}{N^2}\frac{\alpha}{\gamma} + \frac{p}{N}\left(c_5\frac{\beta}{\gamma} + c_6\right)\right]^{-1} \tag{4}$$

In the limit, N must grow with p to maintain efficiency. Notice that the N^2 cannot be readily ignored, even for $N = O(10^3)$, since α is several orders of magnitude greater than γ for many multicomputers. This kind of scalability poses a problem: Memory requirements grow with N^2 and hence eventually N cannot be increased to maintain efficiency. A similar analysis can be done for row distributions.

By contrast, consider a general $P \times Q$ grid of nodes. Assume the ratio Q/P is kept constant as p is increased, i.e., $P = u\sqrt{p}$ and $Q = v\sqrt{p}$, where u and v are constants. Then P/N and Q/N become $u\sqrt{p}/N$ and $v\sqrt{p}/N$, respectively. If $\log(P)$ is ignored, since it is a slowly growing function, N^2 must grow with p in order to maintain efficiency. If $\log(P)$ is not ignored, it can be argued that once P is sufficiently large (e.g. greater than 4) performance will degrade slowly with p.

3.2 Experiments on the Intel Delta

In this section, we discuss results from experiments conducted on the Intel Touchstone Delta that illustrate the scalability of the LU factorization.

The Intel Touchstone Delta system is a distributed-memory, message-passing multicomputer of the Multiple Instruction Multiple Data (MIMD) class [19]. It consists of 520 i860-based nodes, interconnected via a communications network having the topology of a two-dimensional rectangular grid. The interconnection network employs a Mesh Routing Chip (MRC) at each system node. The peak interprocessor communications bandwidth is $\approx$ 30 MBytes/s in each direction. The system supports explicit message-passing, with a latency of $\approx$ 75 microseconds via worm-hole routing using a packet-based protocol. Interconnect blocking is minimized by interleaving packets associated with distinct messages that need to traverse the same interconnect path.

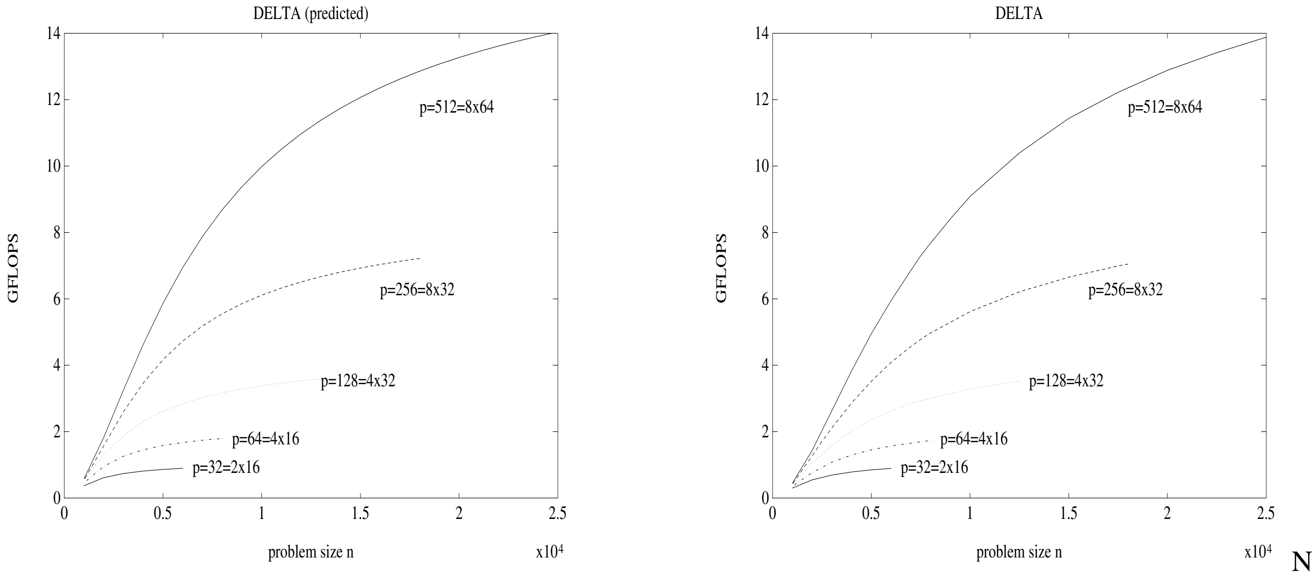


Figure 2: Total predicted (left) and observed (right) performance for various p as a function of the problem size N.

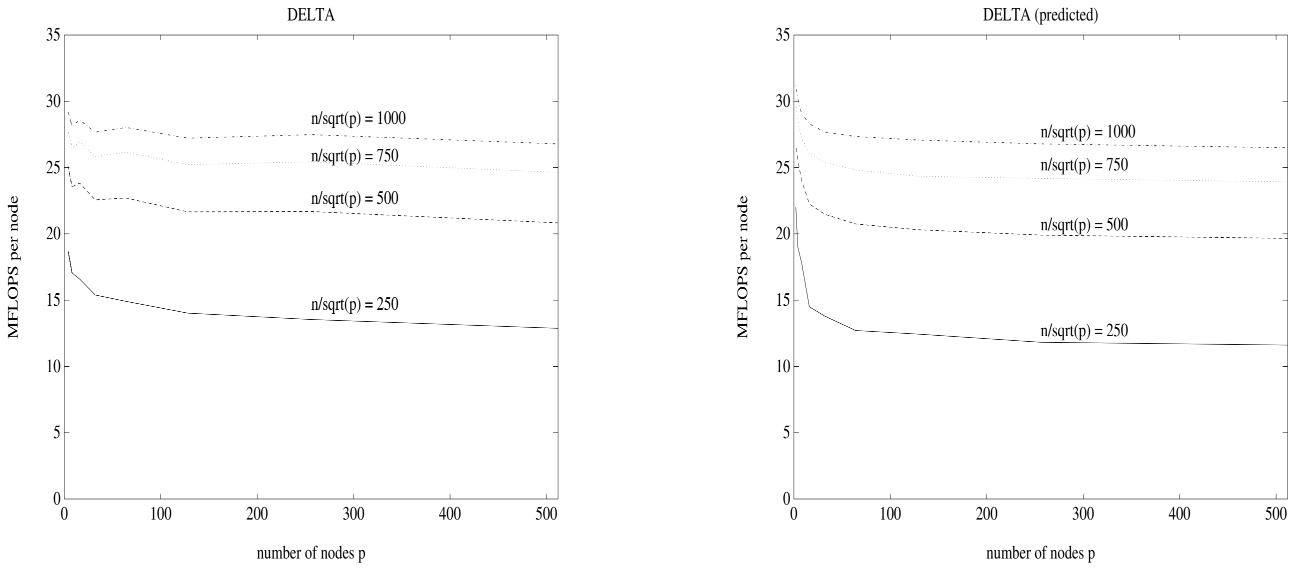


Figure 3: Performance per node predicted (left) and observed (right) as the number of nodes p varies. Different curves correspond to problem sizes increased so that N^2/p (or $N/sqrt(p)$) is constant.

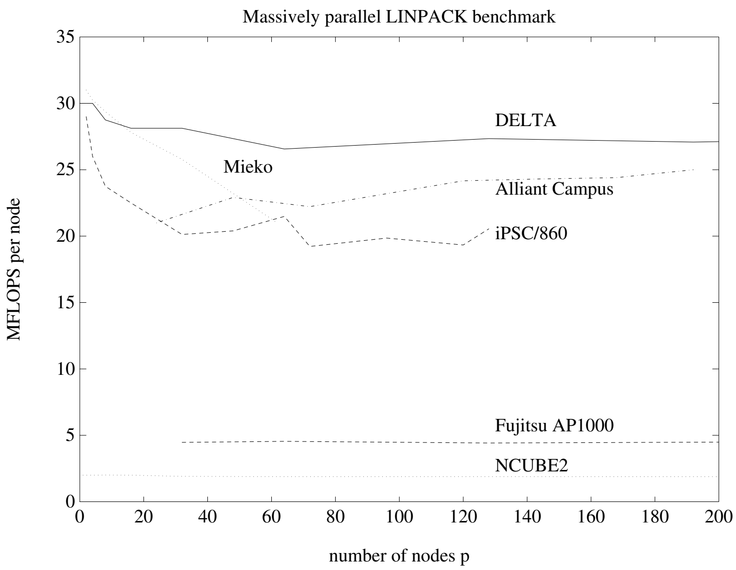


Figure 4: Performance per node attained for the LINPACK benchmark by various parallel architectures as the number of nodes p varies.

There are a number of issues that complicate a direct comparison of our analytical estimates and observed performance. First, certain optimizations can be done to improve the algorithm given in Section 3 [22], details of which go beyond the scope of this paper. Second, the parameter γ is affected by the size of the data being manipulated: computation at different stages involves Level 1, 2, and 3 BLAS, which yield different performance depending on the size of the data being manipulated. Finally, the blocksize r and grid size $P \times Q$ are chosen so the performance of the BLAS is maximized without creating unreasonable idle time due to load inbalance. This leaves us to investigate if the predicted trends can be observed in practice.

In Fig. 2, we report the predicted and observed performance of the LU factorization for different numbers of nodes when the problem size N is varied. For the predicted performance, $\alpha = 100\mu$sec, $\beta = 1\mu$sec (8 Mbytes/sec bandwidth), and $\gamma = 29$nsec (34 MFLOPS per node) where used. (These correspond roughly to what we observed in practice. Communication overhead is somewhat increased by our code.) The grid sizes were experimentally determined to be optimal for large problem sizes. As the problem size increases, performance improves. The results compare favorably with the peak performance that can be attained for this type of problem on the Delta.

The predicted degradation of performance when $N/\sqrt{p}$ is held constant is illustrated in Fig. 3. This trend is also observed in practice, as illustrated in Fig. 3. In these figures, we report efficiency as performance (in MFLOPS) per node.

The LU factorization is at the core of the LINPACK benchmark. This benchmark measures the performance of a given computer while performing a dense linear solve. A typical implementation starts by factoring the matrix, followed by triangular solves. Results from implementations on various parallel architectures are reported in [13]. To illustrate that the predicted trends can be observed on other parallel computers as well, we report performance per node in Fig. 4. While there is a clear incentive to fill the memory with the largest possible problem, thereby automatically increasing N^2 roughly with p, the data made available to us did not in all cases include problem sizes that scaled as nicely as those used for Fig. 3. Although data was available for an NCUBE2 up to size 1024, and for the Fujitsu and Delta up to size 512, we concentrate on the more interesting range of machine sizes in this figure.

Several observations can be made: Both the NCUBE2 and the Fujitsu are based on relatively slow processors.

This decreases the ratios α/γ and α/β, thereby reducing the effects of communication overhead. Moreover, the performance of the BLAS on these machines is less affected by the size of the problem. All other machines are base on the same processor: the Intel i860. The curve for the Mieko follows the predicted trend, except that the last data point (for 62 nodes) is for a much smaller problem size than is required to keep N^2/p constant. At first glance, the efficiency attained by the Alliant appears to improve with the number of nodes, defying the results of our analysis. Moreover, when looking at the raw data, the problem sizes actually grow slower than required by our analysis. This indicates that there is a lower order term that affects performance for small problem sizes. Indeed, it is reportedly due to an inefficient triangular solve algorithm used in this implementation.

4 Programmability

Programmability will be used here to refer to a number of features of the software environment concerned with software maintenance and usage. Programmability covers the flexibility, range of functionality, portablility, and ease of use of some software component. From an application programmer's point of view, the main factor that will determine how easy it is to learn and use the proposed subroutine library will be the interface to the subroutines. Clearly, this interface must pass the appropriate information about the decomposition and layout of the data in memory to the subroutine. This could be done in three ways:

1. by only allowing one type of decomposition for each subroutine so that different subroutines must be called for different decompositions. This avoids having to specify the decomposition in a lengthy argument list, but makes maintaining and porting the subroutine library rather tedious.

2. have a single subroutine handle all possible different decompositions and pass the decomposition information via the argument list. This can result in long argument lists.

3. use an object-oriented approach in which a matrix is actually a data structure containing the data itself (or pointers to it), plus all the information necessary to fully specify the decomposition. This allows a single subroutine to handle all decompositions, and avoids a long argument list. This approach is the most elegant and conceptually simplest for the application programmer. It is rather more difficult to implement than the other two approaches.

The object-oriented approach allows details of the parallel implementation to be hidden at a low level of the software. Ideally, all communication would be hidden below the level of the BLAS routines. In the prototype parallel dense linear algebra library currently under development all interprocessor communication takes place explicitly at the level of the parallel linear algebra routines through calls to a communication library, the LACS routines [3, 16, 14]. Thus, currently the sequential BLAS routines, together with the LACS, are the building blocks used to build higher level library routines, such as LU and QR factorization.

In addition to a set of subroutines for performing matrix computations the proposed library will also contain routines for performing communication tasks. Such tasks will include global changes to the decomposition, such as performing a matrix transpose, and replicating parts of a matrix over groups of nodes. This latter type of communication is similar to the SPREAD routine in Fortran 90 [8], and will allow, for example, row and columns of a matrix to be communicated across across the machine. These LACS could also be given an object-oriented style of interface. In fact, some of the array intrinsic functions of Fortran 90, such as SPREAD, CSHIFT, and EOSHIFT, could be included in the LACS.

Other utility routines will also be provided. One set of assignment routines will be used to initially specify the decomposition, and another set of inquiry routines will provide a means of extracting information about the current decomposition. These inquiry routines will allow application programmers to develop modular subprograms that are fully compatible with our linear algebra library.

5 Conclusions

The square block scattered decomposition (SBS) is a practical and general-purpose way of decomposing dense linear algebra computations. In problems, such as LU factorization, in which rows and/or columns become inactive as the algorithm progresses, the SBS decomposition provides good load balance. At the same time it reduces communication latency since fewer messages need to be sent than in the nonblocked case $(r = 1)$. It is possible to regard each of the blocks as a distinct process, so the SBS decomposition, in effect, overdecomposes the problem. The resultant parallel slackness could then be exploited by overlapping communication and computation. This might be a viable approach on future machines that support multithreading in the operating system kernel, or in hardware. However, on currently available machines the communication latency is probably too high to make it worthwhile, although our general approach should make it easy to exploit overdecomposition in the future.

The LU factorization timings presented in Section 3 show that the SBS decomposition results in scalable and efficient code, attaining a speed of about 14 GFLOPS on the Intel Touchstone Delta system for the largest problem considered.

We propose an object-oriented interface to the library routines, in which the objects are matrices that include pointers to both the matrix data and the decomposition. With this approach all interprocessor communication takes place within the Level 3 BLAS routines, or within the Linear Algebra Communication Subprograms (LACS), which are provided to perform common communication tasks. The user is largely insulated from the details of the parallel implementation, making applications more readily portable, and easier to develop.

Acknowledgements

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 by the California Institute of Technology and Intel Supercomputer Systems Division.

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, 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.

[3] E. Anderson, A. Benzoni, J. Dongarra, S. Moulton, S. Ostroucho v, B. Tourancheau, and R. van de Geijn. Basic Linear Algebra Communication Subprograms. In *Sixth Distributed Memory Computing Conference Proceedings*, pages 287–290. IEEE Computer Society Press, 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] C. C. Ashcraft. A taxonamy of distributed dense LU factorization methods. Engineering Computing and Analysis Technical Report ECA-TR-161, Boeing Computer Services, 1991.

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

[7] E. F. Van de Velde. Data redistribution and concurrency. *Parallel Computing*, 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, 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, I. S. Duff, J. Du Croz, and S. Hammarling. A set of level 3 basic linear algebra subprograms. *ACM TOMS*, 16(1):1–17, March 1990.

[12] 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, Philadelphia, PA, 1990.

[13] Jack J. Dongarra. Performance of various computers using standard linear equations software. Technical report.

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

[15] 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*.

[16] J.J. Dongarra and R.A. van de Geijn. Two dimensional basic linear algebra communication subprograms. LAPACK Working Note 37, Technical Report CS-91-138, University of Tennessee, 1991.

[17] 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.

[18] 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.

[19] S.L. Lillevik. The Touchstone 30 Gigaflop DELTA Prototype. In *Sixth Distributed Memory Computing Conference Procedings*, pages 671–677. IEEE Computer Society Press, 1991.

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

[21] 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 Conference*, pages 328–337. IEEE Press, 1990.

[22] R.A. van de Geijn. Massively parallel LINPACK benchmark on the Intel Touchstone Delta and iPSC/860 systems. In *1991 Annual Users' Conference Proceedings*, Intel Supercomputer Users' Group, Dallas, TX, 1991.