Key Concepts For Parallel Out-Of-Core LU Factorization

Jack J. Dongarra[‡*] Sven Hammarling[*†]
David W. Walker[‡]

March 28, 1996

Abstract

This paper considers key ideas in the design of out-of-core dense *LU* factorization routines. A left-looking variant of the *LU* factorization algorithm is shown to require less I/O to disk than the right-looking variant, and is used to develop a parallel, out-of-core implementation. This implementation makes use of a small library of parallel I/O routines, together with ScaLAPACK and PBLAS routines. Results for runs on an Intel Paragon are presented and interpreted using a simple performance model.

1 Introduction

The in-core solution of dense linear systems typically takes less than one hour on the largest parallel computers, even when the system occupies all of memory. For example, on 1,000 processors of an Intel paragon supercomputer, each with 16 Mbytes of memory, it takes about 22 minutes to factor and solve at 64-bit precision a dense linear system of order 40,000 that fills up all the memory available to applications. This indicates that the processing

[*]Department of Computer Science, University of Tennessee, Knoxville, TN 37996-1301
[†]NAG Ltd., Wilkinson House, Jordan Hill Road, Oxford OX2 8DR, U.K.
[‡]Mathematical Sciences Section, Oak Ridge National Laboratory, Oak Ridge, TN 37831-6367

power of such machines is underutilized in problems that require the solution of a single linear system in the sense that much larger systems could be solved before the run time became prohibitively large. In the absence of substantial increases in the ratio of memory to processing power it is natural to develop out-of-core solvers to tackle very large linear systems. These types of large linear system arise, for example, in three-dimensional electromagnetic scattering problems and in fluid flow past complex objects [10, 11].

This paper presents a prototype for the design of a parallel software library for the out-of-core solution of dense linear systems. In section 2, we consider left- and right-looking, out-of-core parallel LU factorization routines and propose a hybrid version that balances the degree of parallelism with the amount of I/O. In section 4 different approaches to parallel I/O are discussed. Section 5 outlines the main components of a library of routines for performing I/O on dense matrices. A complete parallel, out-of-core LU factorization routine is described in section 6. This algorithm is implemented in terms of the BLACS [9], PBLAS [3], and ScaLAPACK [2] routines. Section 7 presents some preliminary performance results on the Intel Paragon. A summary and conclusions are presented in section 8.

2 Sequential Out-Of-Core LU Factorization

Let us consider the decomposition of the matrix A into its LU factorization with the matrix partitioned in the following way. Let us suppose that we have factored A as $A = LU$. We write the factors in block-partitioned form and observe the consequences.

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix} = \begin{pmatrix} L_{11} & & \\ L_{21} & L_{22} & \\ L_{31} & L_{32} & L_{33} \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} & U_{13} \\ & U_{22} & U_{23} \\ & & U_{33} \end{pmatrix}$$

Multiplying L and U together and equating terms with A, we have

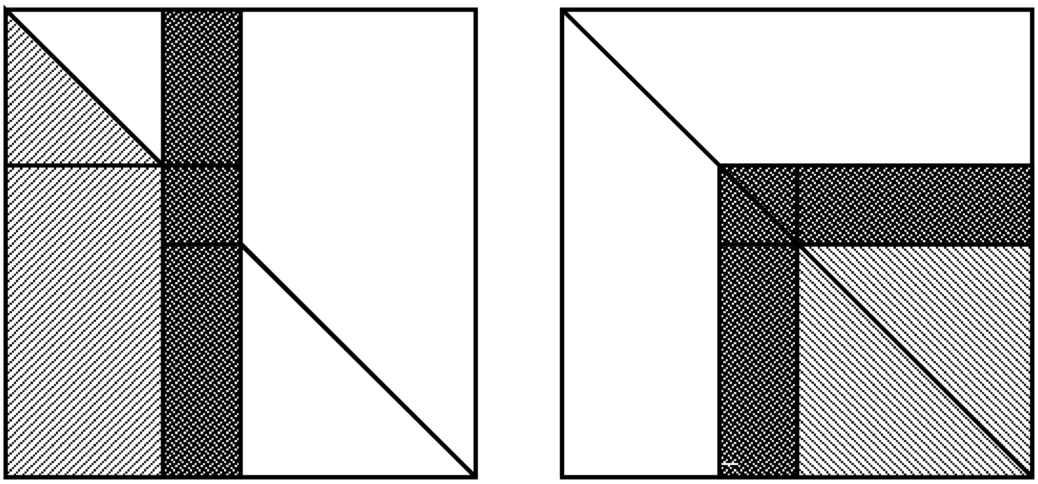

Figure 1: Memory access patterns for variants of LU decomposition. The shaded parts indicate the matrix elements accessed in forming a block row or column, and the darker shading indicates the block row or column being modified.

$$
\begin{array}{lll}
A_{11} = L_{11}U_{11}, & A_{12} = L_{11}U_{12}, & A_{13} = L_{11}U_{13}, \\
A_{12} = L_{21}U_{11}, & A_{22} = L_{21}U_{12} + L_{22}U_{22}, & A_{23} = L_{21}U_{13} + L_{22}U_{23}, \\
A_{31} = L_{31}U_{11}, & A_{32} = L_{31}U_{12} + L_{32}U_{22}, & A_{33} = L_{31}U_{13} + L_{32}U_{23} + L_{33}U_{33}.
\end{array}
$$

With these simple relationships we can develop variants by postponing the formation of certain components and also by manipulating the order in which they are formed. A crucial factor for performance is the choice of the *blocksize, k* (i.e., the column width) of the second block column. A blocksize of 1 will produce matrix-vector algorithms, while a blocksize of $k > 1$ will produce matrix-matrix algorithms. Machine-dependent parameters such as cache size, number of vector registers, and memory bandwidth will dictate the best choice for the blocksize.

Two natural variants occur: right-looking and left-looking. (There are several other variants possible, we examine only two here.) The terms right and left refer to the regions of data access, as shown in Figure 1.

The left-looking variant computes one block column at a time, using previously computed columns. The right-looking variant (the familiar recursive algorithm) computes a block row and column at each step and uses them to update the trailing submatrix. These variants have been called the *i,j,k variants* owing to the arrangement of loops in the algorithm. For a more complete discussion of the different variants, see [8, 13].

We now develop these block variants of LU factorization with partial pivoting.

2.1 Right-Looking Algorithm

Suppose that a partial factorization of A has been obtained so that the first k columns of L and the first k rows of U have been evaluated. Then we may write the partial factorization in block partitioned form, with square blocks along the leading diagonal, as

$$PA = \begin{pmatrix} L_{11} & & \\ L_{21} & I & \\ L_{31} & 0 & I \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} & U_{13} \\ & \hat{A}_{22} & \hat{A}_{23} \\ & \hat{A}_{32} & \hat{A}_{33} \end{pmatrix}, \tag{1}$$

where L_{11} and U_{11} are $k \times k$ matrices, and P is a permutation matrix representing the effects of pivoting. Pivoting is performed to improve the numerical stability of the algorithm and involves the interchange of matrix rows. The blocks labeled $\hat{A}_{ij}$ in Eq. 1 are the updated portion of A that has not yet been factored, and will be referred to as the *active submatrix*.

We next advance the factorization by evaluating the next block column of L and the next block row of U, so that

$$\begin{pmatrix} I & \\ & P_2 \end{pmatrix} PA = \begin{pmatrix} L_{11} & & \\ L_{21} & L_{22} & \\ L_{31} & L_{32} & I \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} & U_{13} \\ & U_{22} & U_{23} \\ & & \hat{A}_{33} \end{pmatrix}. \tag{2}$$

where P_2 is a permutation matrix of order $M-k$. Comparing Eqs. 1 and 2 we see that the factorization is advanced by first factoring the first block column of the active submatrix which will be referred to as the *current column*,

$$P_2 \begin{pmatrix} \hat{A}_{22} \\ \hat{A}_{32} \end{pmatrix} = \begin{pmatrix} L_{22} \\ L_{32} \end{pmatrix} U_{22} \tag{3}$$

This gives the next block column of L. We then pivot the active submatrix to the right of the current column and the partial L matrix to the left of the current column,

$$\begin{pmatrix} \hat{A}_{23} \\ \hat{A}_{33} \end{pmatrix} \Leftarrow P_2 \begin{pmatrix} \hat{A}_{23} \\ \hat{A}_{33} \end{pmatrix}, \quad \begin{pmatrix} L_{21} \\ L_{31} \end{pmatrix} \Leftarrow P_2 \begin{pmatrix} L_{21} \\ L_{31} \end{pmatrix} \tag{4}$$

and solve the triangular system

$$U_{23} = {L_{22}}^{-1} \hat{A}_{23} \tag{5}$$

to complete the next block row of U. Finally, a matrix-matrix product is performed to update $\hat{A}_{33}$,

$$\hat{A}_{33} \Leftarrow \hat{A}_{33} - L_{32} U_{23}. \tag{6}$$

Now, one simply needs to relabel the blocks to advance to the next block step.

The main advantage of the block partitioned form of the LU factorization algorithm is that the updating of $\hat{A}_{33}$ (see Eq. 6) involves a matrix-matrix operation if the block size is greater than 1. Matrix-matrix operations generally perform more efficiently than matrix-vector operations on high performance computers. However, if the block size is equal to 1, then a matrix-vector operation is used to perform an outer product — generally the least efficient of the Level 2 BLAS [7] since it updates the whole submatrix.

Note that the original array A may be used to store the factorization, since the L is unit lower triangular and U is upper triangular. Of course, in this and all of the other versions of LU factorization, the additional zeros and ones appearing in the representation do not need to be stored explicitly.

We now derive the cost for performing I/O to and from disk for the block-partitioned, right-looking LU factorization of an $M \times M$ matrix A with a block size of n_b. For clarity assume M is exactly divisible by n_b. The factorization proceeds in M/n_b steps which we shall index $k = 0, 1 \ldots, M/n_b - 1$.

For some general step k, the active submatrix is the $M_k \times M_k$ matrix in the lower right corner of A, where $M_k = M - kn_b$. In step k it is necessary to both read and write all of the active submatrix, so the total I/O cost for the right-looking algorithm is

$$(R+W)\sum_{k=0}^{M/n_b-1}(M-kn_b)^2 = \frac{M^3}{3n_b}\left(1+O(n_b/M)\right)(R+W) \tag{7}$$

where R and W are the times to read and write one matrix element, respectively, and we assume there is no startup cost when doing I/O.

2.2 Left-Looking Algorithm

As we shall see, from the standpoint of data access, the left-looking variant is better than the right-looking variant. To begin, we assume that

$$PA = \begin{pmatrix} L_{11} & & \\ L_{21} & I & \\ L_{31} & 0 & I \end{pmatrix}\begin{pmatrix} U_{11} & A_{12} & A_{13} \\ 0 & A_{22} & A_{23} \\ 0 & A_{32} & A_{33} \end{pmatrix}. \tag{8}$$

and that we wish to advance the factorization to the form

$$\begin{pmatrix} I & \\ & P_2 \end{pmatrix} PA = \begin{pmatrix} L_{11} & & \\ L_{21} & L_{22} & \\ L_{31} & L_{32} & I \end{pmatrix}\begin{pmatrix} U_{11} & U_{12} & A_{13} \\ 0 & U_{22} & A_{23} \\ 0 & 0 & A_{33} \end{pmatrix}. \tag{9}$$

Comparing Eqs. 8 and 9 we see that the factorization is advanced by first solving the triangular system

$$U_{12} = L_{11}^{-1}A_{12} \tag{10}$$

and then performing a matrix-matrix product to update the rest of the middle block column of U,

$$\begin{pmatrix} \hat{A}_{22} \\ \hat{A}_{32} \end{pmatrix} \Leftarrow \begin{pmatrix} A_{22} \\ A_{32} \end{pmatrix} - \begin{pmatrix} L_{21} \\ L_{31} \end{pmatrix} U_{12}. \tag{11}$$

Next we perform the factorization

$$P_2 \begin{pmatrix} \hat{A}_{22} \\ \hat{A}_{32} \end{pmatrix} = \begin{pmatrix} L_{22} \\ L_{32} \end{pmatrix} U_{22} \tag{12}$$

and lastly the pivoting

$$\begin{pmatrix} A_{23} \\ A_{33} \end{pmatrix} \Leftarrow P_2 \begin{pmatrix} A_{23} \\ A_{33} \end{pmatrix} \text{ and } \begin{pmatrix} L_{21} \\ L_{31} \end{pmatrix} \Leftarrow P_2 \begin{pmatrix} L_{21} \\ L_{31} \end{pmatrix}. \tag{13}$$

Observe that data accesses all occur to the left of the block column being updated. Moreover, the only write access occurs within this block column. Matrix elements to the right are referenced only for pivoting purposes, and even this procedure may be postponed until needed with a simple rearrangement of the above operations.

In evaluating the I/O cost for the left-looking out-of-core LU factorization algorithm two variants of the left-looking algorithm will be considered. In the first we always store the matrix on disk in unpivoted form at all intermediate phases of the algorithm, writing out the whole matrix in pivoted form only in the last step of the algorithm. In this case pivoting has to be done "on the fly" when matrix blocks are read in from disk. In the second version of the algorithm the matrix is stored on disk in pivoted form.

Consider the version in which the matrix is stored in unpivoted form. Whenever a block is read in the whole $M \times n_b$ block must be read so that it can be pivoted. Upon completion of a step the newly-factored block is the only block that is written to disk, except in the last step in which we write out all blocks in pivoted form so that the final matrix stored on disk is pivoted (although in some cases these writes may be omitted if an unpivoted matrix is called for – the pivots can always be applied later since they are stored in the pivot vector). At some general step k of the algorithm the I/O cost is

$$(R+W)Mn_b + RMn_bk \tag{14}$$

where the first term corresponds to reading and writing the block to be factored in this step and the second term to reading in the blocks to the left. Summing over k and adding in the time to write out all pivoted blocks in the last step, the total cost for this version of the left-looking algorithm is

$$\frac{M^3}{2n_b}\left(1 + O(n_b/M)\right)R + 2M^2\left(1 + O(n_b/M)\right)W \tag{15}$$

Thus, to order n_b/M the time to do the writes can be ignored. If we assume that reads and writes take approximately the same time (i.e., $R \approx W$), then comparison with Eq. 7 shows that this version of the left-looking algorithm should perform less I/O than the right-looking algorithm.

Now consider the version of the left-looking algorithm in which blocks are always stored on disk in pivoted form. In this case it is no longer necessary to read in all rows of an $M \times n_b$ block, but it is necessary to write out partial blocks in each step. This is because the pivoting performed in the factorization of the block column must also be applied to the blocks to the left, which must then be written to disk. In some general step k all of the block to be updated must be read in and written out. The parts of the blocks to the left that must be read in form a stepped trapeziodal shape (see Figure 2(a)), while the parts of the blocks to the left that must be written out after applying the pivots for this step form a rectangle (see Figure 2(b)). Thus for step $k > 0$ the I/O cost is

$$(R + W)Mn_b + Rn_b\sum_{i=0}^{k-1}(M - in_b) + Wn_b(M - kn_b)k \tag{16}$$

and for step $k = 0$ the I/O cost is $(R + W)Mn_b$. Thus, the total I/O cost is

$$\frac{M^3}{3n_b}\left(1 + O(n_b/M)\right)R + \frac{M^3}{6n_b}\left(1 + O(n_b/M)\right)W \tag{17}$$

It is interesting to note that if reads and writes take the same time the two left-looking versions of the algorithm have the same I/O cost, and they both have a lower I/O cost than the right-looking algorithm. We therefore expect a left-looking algorithm to be better than a right-looking algorithm for out-of-core LU factorization.

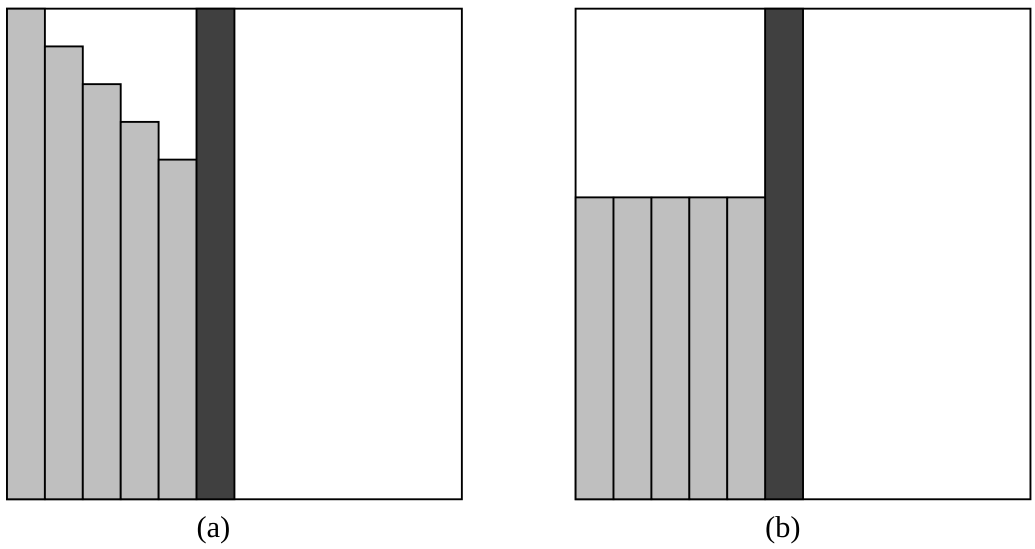

Figure 2: This figure pertains to the left-looking LU factorization algorithm that stores the matrix in pivoted form. (a) The shaded blocks show the block columns read from disk in step $k = 5$. The dark shaded block is the block being updated in this step. (b) The shaded blocks show the block columns written to disk in step $k = 5$.

3 Implementation of the Left-Looking Algorithm

In this section the implementation of the sequential, left-looking, out-of-core LU factorization routine will be discussed. As we shall see in Section 6, once the sequential version has been implemented it is a relatively easy task to parallelize it using the BLACS, PBLAS, and ScaLAPACK, and the parallel out-of-core routines described in Section 5.

In the out-of-core algorithm only two block columns of width n_b may be in-core at any time. One of these is the block column being updated and factored which we shall refer to as the *active block*. The other is one of the block columns lying to the left of the active block column which we shall refer to as a *temporary block*. As we saw in Section 2.2, the three main computational tasks in a step of the left-looking algorithm are a triangular solve (Eq. 10), a matrix-matrix multiplication (Eq. 11), and an LU factorization (Eq. 12). In the out-of-core algorithm the triangular solve and matrix-matrix multiplication steps are intermingled so that a temporary block can play its part in both of these operations but be read only once. To clarify this, consider the role that block column i plays in the factorization of block column k (where $i < k$). In Figure 3, the first i rows of block column i play no role in factoring block column k. The lower triangular portion of the next n_b rows of block column i are labeled T_0, and the next $k - i - n_b$ rows are labeled T_1. The last $M - k$ rows are labeled D. The corresponding portions of block column k are labeled C_0, C_1, and E. Then the part played by block column i in factoring block column k can be expressed in the following three operations,

$$
\begin{aligned}
C_0 &\leftarrow T_0^{-1} C_0 && (18)\\
C_1 &\leftarrow C_1 - T_1 C_0 && (19)\\
E &\leftarrow E - D C_0 && (20)
\end{aligned}
$$

where in Eqs. 19 and 20 we use the C_0 given by Eq. 18. It should be noted that Eqs. 19 and 20 can be combined in a single matrix-matrix multiplication operation

$$
\begin{pmatrix} C_1 \\ E \end{pmatrix} \leftarrow \begin{pmatrix} C_1 \\ E \end{pmatrix} - \begin{pmatrix} T_1 \\ D \end{pmatrix} C_0. \qquad (21)
$$

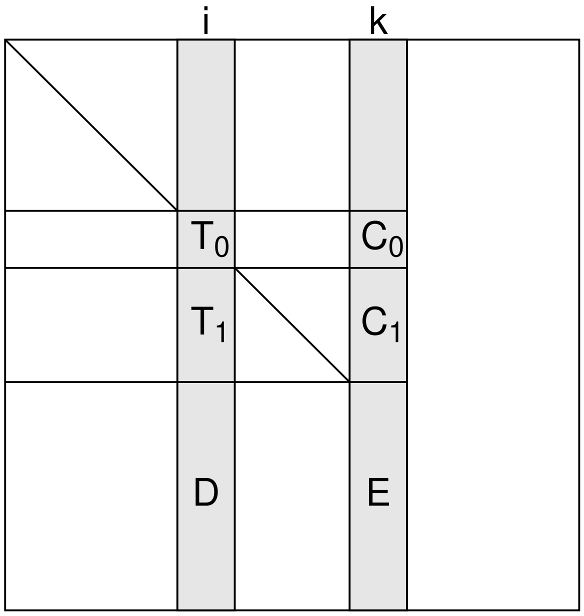

Figure 3: Partitioning of temporary block i and active block k.

In updating block column k, the out-of-core algorithm sweeps over all block columns to the left of block column k and performs for each the triangular solve in Eq. 18 and the matrix-matrix multiplication in Eq. 21. After all the block columns to the left of the block have been processed in this way using the Level 3 BLAS routines _TRSM and _GEMM [6], the matrix E is then factored using the LAPACK routine _GETRF [1].

If the matrix is stored on disk without applying the pivots to it, then whenever a block column is read in the pivots found up to that point must be applied to it using _LASWP, an LAPACK auxiliary routine. Also after updating and factoring the active block, the pivots must be applied to it in reverse order to undo the effect of pivoting before storing the block column to disk. In this version of the left-looking algorithm complete block columns are always read or written. In the version of the algorithm in which the matrix is stored on disk in pivoted form it is necessary to read in only those parts of the temporary blocks that play a role in the computation. When a partial temporary block is read in the pivots found when factoring E in the previous step must be applied before using it, and it must then be written

```
for (each block column, k=0,1,...,M/n_b-1)
   read block column k into active block
   _LASWP : apply pivots to active block
   go to start of file
   for (each block column to left, i=0,1,...k-1)
      read block column i into temporary block
      _LASWP : apply pivots to temporary block
      _TRSM  : triangular solve
      _GEMM  : matrix multiply
   end for
   _GETRF : factor matrix E
   _LASWP : unpivot active block
   write active block
end for
```

Figure 4: Pseudocode for out-of-core, left-looking LU factorization algorithm that leaves matrix in unpivoted form.

back out to disk.

In Figure 4 the pseudocode is presented for the version of the left-looking algorithm in which the matrix is stored in unpivoted form. Since a vector of pivot information is maintained in-core, the factored matrix can always be read in later to be pivoted. It has been assumed in Figure 4 that the matrix is $M \times M$ and that M is divisible by the block size n_b. However, the general case is scarcely more complicated. It should be noted that it is necessary to position the file pointer (at the start of the file) only once in each pass through the outer loop.

4 Approaches To Parallel I/O

Our discussion of parallel I/O for dense matrices assumes that in-core matrices are distributed over processes using a block-cyclic data distribution as in ScaLAPACK [4, 2]. Processes are viewed as being laid out with a two-dimensional logical topology, forming a $P \times Q$ process mesh. Our approach to parallel I/O for dense matrices hinges on the number of file pointers, and on which processes have access to the file pointers. We divide parallel I/O modes into two broad classes

1. There is one file pointer into the disk file. In this case some of the possibilities are

   (a) Only one process has access to the file pointer. Thus only that process can do I/O to the file, and has to scatter to, or gather from, the other processes when reading or writing the file.

   (b) All processes in a group have individual access to the file pointer. Synchronization is required if the order in which data are written to, or read from, the file is important.

   (c) All processes in a group have collective access to the file pointer permitting collective I/O operations in which all processes can read the same data from the file, or collectively write to the file in such a way that the data from exactly one of the processes is actually written to the file.

2. Each process in a group has its own file pointer. We consider here two main possibilities

   (a) The file pointers can all access a global file space. In this case we refer to the file as a "shared file."

   (b) each file pointer can only access its own local file space. This file space is physically and logically contiguous. In this case we refer to the file as a "distributed file."

Modes 1(a) and 1(b) correspond to the case in which there is no parallel I/O system, and all I/O is bound to be sequential. Modes 1(c), 2(a) and 2(b) corresponds to different ways of doing parallel I/O. The shared file mode is the most general since it means a file can be written using one particular process grid and block size and read later using a different process grid and block size. A distributed file can only be read using the same process grid and block size that it was written with. However, a major drawback of a shared file is that, in general, each process can only read and write n_b contiguous elements at a time. This results in very poor performance unless block sizes are very large or unless the process grid is chosen to be $1 \times Q$ (for Fortran codes) so that each column of the matrix lies in one process. The potential for poor performance arises because most I/O systems work best when reading large blocks. Furthermore, if only a small amount of data is written at a time systems such as the Intel Paragon will not stripe the data across disks so I/O is essentially sequentialized.

5 Parallel I/O Routines For Dense Matrices

We propose a prototype library of Basic Linear Algebra Parallel I/O Subprograms (BLAPIOS) for dense matrices. As discussed in Section 3, we would like the BLAPIOS to be compatible with any future standard for parallel I/O that emerges. Thus, we describe only the high-level functionality of the BLAPIOS, and defer specifying the detailed semantics and syntax. A similar approach has been taken by Toledo and Gustavson in the Matrix Input-Output Subroutines (MIOS) which forms part of the SOLAR library for out-of-core dense matrix computations [15].

Before describing the BLAPIOS we shall consider the fundamental I/O operation supported by the BLAPIOS in which a rectangular array of data is read from (written to) the out-of-core file into (from) a given in-core array. Suppose the data in the out-of-core file and the in-core array are represented by the index ranges $(k : k+m-1, \ell : \ell+n-1)$, and $(i : i+m-1, j+n-1)$, respectively, as shown in Figure 5. As in the PBLAS and ScaLAPACK libraries, submatrices are regarded as global entities and are referenced by global indices.

For a shared file the indices k and ℓ can refer to any element in the out-of-core file. However, for a distributed file the submatrix referenced in the out-of-core file must have the same data distribution as that in the in-core array. This is because both the out-of-core distributed file and the in-core array are distributed data objects. An example of compatible and incompatible data distributions for a distributed file and an in-core matrix are shown in Figure 6.

The routines comprising the BLAPIOS library are arranged in three groups.

- Routines for opening and closing files, and for manipulating file pointers.

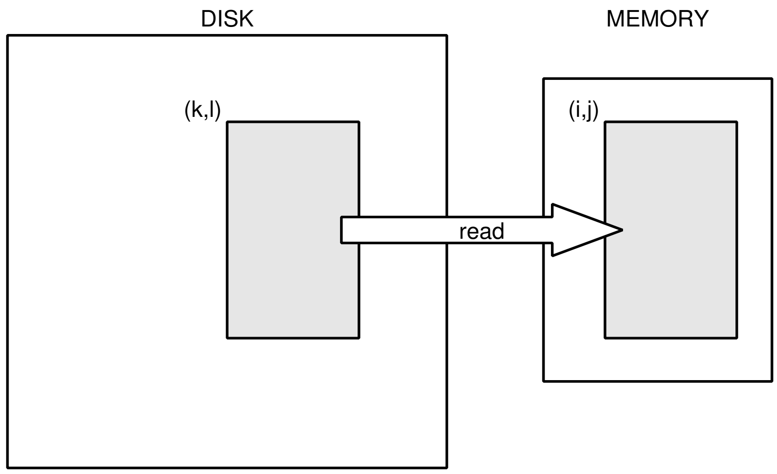

Figure 5: Fundamental I/O operation for matrices.

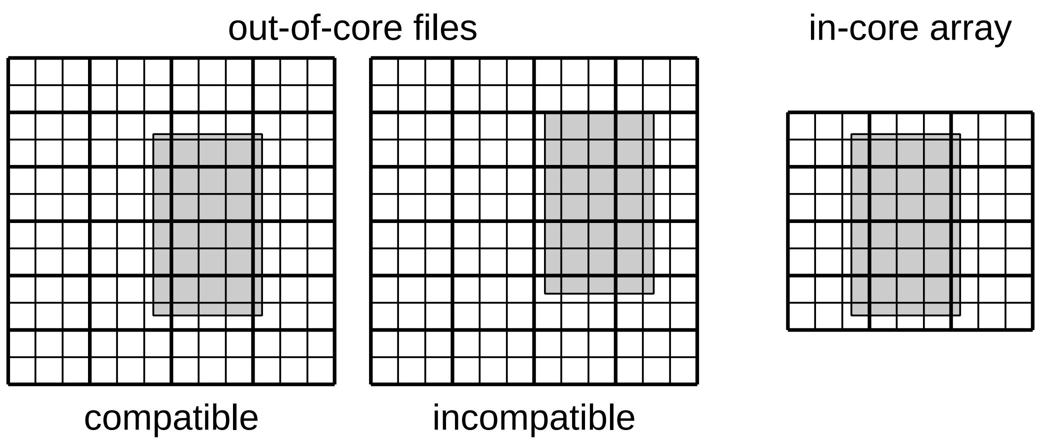

Figure 6: On the left we show two submatrices of a distributed file. On the right is an in-core array. Both the distributed file and the in-core array are distributed over a 2×3 mesh of processes. The smaller squares represent $n_b \times n_b$ blocks of elements. The distribution of the submatrix in the lefthand distributed file is compatible with that in the in-core array, while the distribution of the submatrix in the righthand distributed file is not.

- Routines for reading and writing.
- Auxiliary routines.

We shall now present the functionality of each of these routines.

5.1 File Management Routines

The BLAPIOS contain the following routines for handling shared and distributed files.

POPEN. Opens a file.

PCLOSE. Closes a file.

P_LSEEK. Independently positions the file pointer to a specific location in the file.

P_ASEEK. Positions the file pointers according to an explicit alignment. For a distributed file the alignment must be compatible with the data distributions of the out-of-core file and the in-core array.

P_GSEEK. Positions the file pointers according to an implicit alignment obtained by applying a given data distribution over the out-of-core file. For a distributed file, the data distribution applied must be that of the distributed file. This is useful when it is known that a subsequent I/O operation will refer to a compatibly aligned in-core array.

5.2 I/O Routines

The BLAPIOS provide the following blocking and nonblocking routines for reading and writing submatrices of an out-of-core file. The nonblocking routines permit the possibility of overlapping I/O to disk with computation and interprocess communication.

P_READ. Reads a submatrix into specified location of a matrix, and leaves the file pointer for each process at the next data element for the process. This is a blocking call.

P_WRITE. Writes a submatrix from specified location of a matrix, and leaves the file pointer for each process at the next data element for the process. This is a blocking call.

P_IREAD. Reads a submatrix into specified location of a matrix, and leaves the file pointer for each process at the next data element for the process. This is a nonblocking call.

P_IWRITE. Writes a submatrix from specified location of a matrix, and leaves the file pointer for each process at the next data element for the process. This is a nonblocking call.

PIOTEST. Tests if a nonblocking parallel I/O call has completed.

PIOWAIT. Blocks until a nonblocking parallel I/O call has completed.

5.3 Auxiliary Routines

The BLAPIOS include the following auxiliary routines.

P_STOD. Converts a shared file to a distributed file.

P_DTOS. Converts a distributed file to a shared file.

P_RANM. Produces a random out-of-core file using a parallel random number generator.

5.4 Implementation Issues

The BLAPIOS outlined above have been implemented on the Intel Paragon using Intel's Parallel File System (PFS). In these PFS-BLAPIOS a distributed file is implemented by having each process access its own distinct file, though it could also have been implemented by partitioning a single file into contiguous chunks and assigning each process one chunk. For both shared and distributed modes the M_ASYNC I/O mode of PFS is used. Although one might expect the best performance on a particular platform to come from implementing the BLAPIOS directly on top of the native parallel I/O system, there are also distinct advantages to being able to implement them on top of a portable parallel I/O system. Parallel I/O is an area of much active research (see, for example, [12] and the parallel I/O archive at http://www.cs.dartmouth.edu/pario.html

for more information.) Although there is currently no generally accepted parallel I/O standard, MPI-IO, the proposed extensions to MPI [14] for performing parallel I/O, is a strong contender [5]. We shall, therefore, briefly consider how the BLAPIOS might be implemented on top of MPI-IO.

MPI-IO contains routines for collective and independent I/O operations. All the I/O operations in the BLAPIOS are independent. MPI-IO partitions a file using filetypes, which are an extension of MPI datatypes. Each process in a given group (specified by an MPI communicator) creates a filetype that picks out just the data assigned to it. A routine for creating a filetype for block-cyclicly distributed matrices is provided by MPI-IO. This filetype, together with MPI-IO's absolute offset mode, can be used to create and access the equivalent of a BLAPIOS shared file. A BLAPIOS distributed file can be handled by creating a datatype that divides the file into contiguous segments with one segment being assigned to each process. In this case MPI-IO's relative offset mode would be used to access data.

In MPI-IO the filetype and communicator are specified as input arguments when a file is opened. This is somewhat more restrictive than access to a shared file using the BLAPIOS in which the partitioning is determined dynamically by the distribution of the in-core matrix being read from or written to. The usefullness of dynamic partitioning (or alignment) is apparent when performing the LU factorization of A, an $M \times N$ matrix with $N > M$. In this case there are two phases to the computation: first the LU factorization of the first M columns is found (call this matrix B), and then the transformations are applied to the remaining $N - M$ columns (call this matrix C). It is natural, and convenient, in performing the second phase of the algorithm to treat matrices B and C as unrelated matrices with independent partitionings. However, complications can arise if the number of columns spanning the process grid, Qn_b, does not exactly divide M, so that C begins in the middle of a block. If we are dealing with a shared file the BLAPIOS routine P_ASEEK can be used to dynamically partition C so it starts at the beginning of a block. For a distributed file, which has a fixed partitioning, we have to offset the in-core matrix involved in I/O operations so that it is aligned with the partitioning. To make the BLAPIOS compatible with MPI-IO we need to either permit multiple alignments for a file in MPI-IO, or else permit only fixed alignments for shared files in the BLAPIOS.

6 A Parallel Algorithm

Although in section 2 we saw that the left-looking LU factorization routine has a lower I/O cost that the right-looking variant, the left-looking algorithm has less inherent parallelism since it acts only on single blocks. We therefore propose a hybrid parallel algorithm in which a single block actually spans several widths of the process grid, say n_g. In effect, the matrix is now blocked at two levels. It is divided into blocks of size n_b elements, which are distributed cyclicly over the process grid, but we apply the left-looking algorithm to "superblocks" of width $n_b n_g Q$ columns where the process grid is assumed to be of size $P \times Q$. If n_g is chosen large enough we have a pure right-looking algorithm, and if n_g and Q are both 1 we essentially recover the pure left-looking algorithm. Within a superblock we use a right-looking LU factorization algorithm (P_GETRF) to get good parallelism, but at the superblock level we employ a left-looking algorithm to control I/O costs. The parameter n_g can be used to trade off parallelism and I/O cost.

In Figure 7 we show an example for a 2×3 process grid, and $n_g = 2$. For clarity we consider here a matrix consisting of only four column superblocks, though in a "real" application we would expect the number to be much larger. In Figure 7 the first two superblocks have been factored, while the third and fourth superblocks have not yet been changed. We now consider the next stage of the algorithm in which the third superblock, for which the data distribution is shown explicitly, is factored. Note that each of the small numbered squares is actually an $n_b \times n_b$ block, with the numbering indicating the position in the process grid to which it is assigned. At the end of this stage of the algorithm the first three superblocks will have been factored, and the fourth will still be unchanged. In the following we shall refer to the superblock being factored as the *active superblock*.

The parallel implementation closely follows the sequential implementation presented in Section 3. Block columns are read and written using the routines P_READ and P_WRITE. The file pointer is positioned with P_GSEEK. These routines are part of the BLAPIO library introduced in Section 5. The triangular solve and matrix multiplication are done using PBLAS routines. Pivoting is performed by the ScaLAPACK auxiliary routine P_LAPIV, while the factorization is done by the ScaLAPACK routine P_GETRF. Since all these routines reference matrices as global data structures, parallelization of the sequential algorithm is almost trivial. Pseudocode for the parallel version

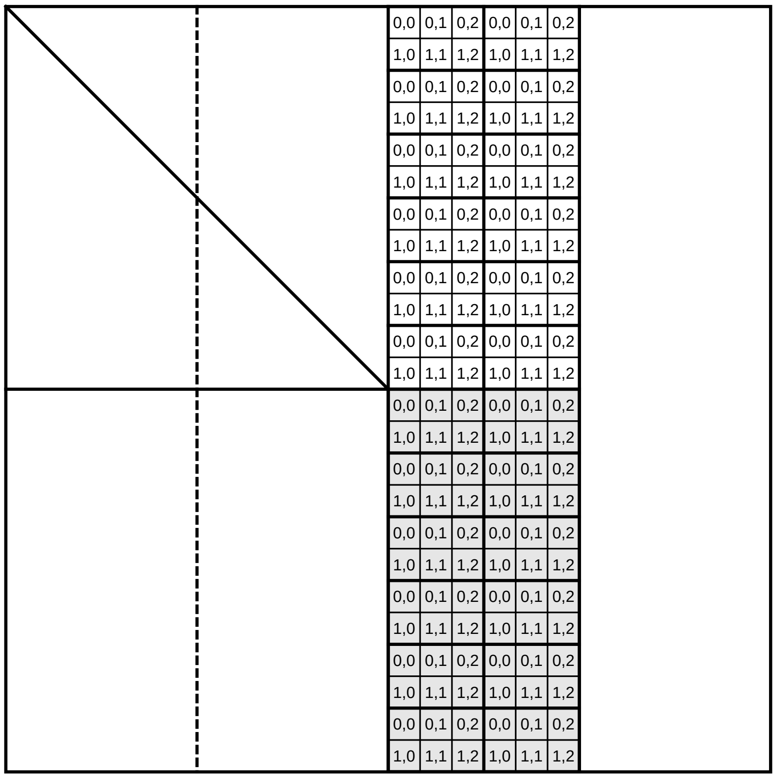

Figure 7: Schematic view of the parallel hybrid out-of-core algorithm for the case $P \times Q = 2 \times 3$ and $n_g = 2$.

```
P_GSEEK : go to start of file
for (each superblock column, k=0,1,...,M/n_b-1)
   P_READ  : read superblock column k into active superblock
   P_LAPIV : apply pivots to active superblock
   P_GSEEK  : go to start of file
   for (each superblock column to left, i=0,1,...k-1)
      P_READ  : read superblock column i into temporary superblock
      P_LAPIV : apply pivots to temporary superblock
      P_TRSM  : triangular solve
      P_GEMM  : matrix multiply
   end for
   P_GETRF : factor lower portion of active superblock
   P_LAPIV : unpivot active superblock
   P_WRITE : write active superblock
end for
```

Figure 8: Pseudocode for parallel, out-of-core, left-looking LU factorization algorithm that leaves matrix in unpivoted form.

is given in Figure 8.

7 Performance Results

In this section some preliminary performance results are presented for the parallel left-looking LU factorization algorithm running on an Intel Paragon concurrent computer. These results are intended to illustrate a few general points about the performance of the algorithms used, and do not constitute a detailed performance study. In the work presented here we were constrained by difficulties encountered in getting exclusive access to the Paragon for sufficiently long periods. In addition we found that the parallel file system of the Paragon to which we had access was close to full much of the time. We hope to overcome these problems in the future and undertake a detailed performance study in future work. All the runs were made in exclusive use mode, i.e., with logins disabled to prevent other users accessing the system.

This was done because the performance of PFS is affected by the load on the service nodes, even if other users are just editing or compiling.

The first runs were done using the version of the algorithm that maintains the partially factored matrix in unpivoted form throughout the algorithm. Timing results are shown for 4×4 and 8×8 process meshes in Tables 1 and 2 for a distributed out-of-core matrix. In these cases we say that the matrix was both logically and physically distributed because each processor opens a separate file. As expected for this version of the algorithm, the time spent writting to PFS is much less than the time spent reading. However, the most striking aspect of the timings is the fact that pivoting dominates. The large amount of time spent pivoting arises because each time a superblock is read in all the pivots evaluated so far must be applied to it. For a sequential algorithm (i.e., $P = Q = n_g = 1$), a total of $M^3/(3n_b^2)$ superblocks of width n_b elements must be pivoted. Thus, pivoting entails $M^3/(3n_b)$ exchanges of elements, which is of the same order as the I/O cost. In the parallel case, we must replace n_b by the width of a superblock, Qn_gn_b. Thus, in order for the version of the algorithm that stores the matrix in unpivoted form to be asymptotically faster than the version that stores the matrix in pivoted form we require

$$\frac{W}{6} < \frac{R}{6} + \frac{P}{3}, \tag{22}$$

where W and R are the costs of writing and reading an element, respectively, and P is the cost of pivoting an element.

In general, there is no reason why writing should be substantially faster then reading, so we would not expect Eq. 22 to hold. Thus, the version of the algorithm that stores the matrix in pivoted form is expected to be faster. This is borne out by the timings presented in Table 3 for an 8×8 process mesh. These timings are directly comparable with those of Table 2, and show that the version of the algorithm that stores the matrix in pivoted form is faster by 10-15%. Note that the time for writing is slightly more than half the time for reading, suggesting that it takes slightly longer to write a superblock than to read it.

We next attempted to investigate the effect of varying the width of the superblock by increasing n_g from 2 to 10. The results are shown in Table 4. A problem will fit in core if the memory required in each process to hold two

Task	5,000	8,000	10,000
Read	67.32	196.73	325.16
Write	9.21	24.39	31.97
Pivot	156.55	538.38	1006.03
Triangular solve	52.88	139.14	219.75
Matrix multiply	115.21	483.37	955.33
Factorization	29.98	65.32	95.76
Total	427.74	1557.16	2802.84

Table 1: Timings in seconds for the main phases of out-of-core LU factorization of $M \times M$ matrices. Results are shown for $M = 5000$, 8000 and 10000. In all cases $n_b = 50$, $n_g = 2$, $P = 4$, and $Q = 4$. The version of the algorithm that stores the matrix in unpivoted form and performs pivoting on the fly was used. The out-of-core matrix was physically and logically distributed.

Task	5,000	8,000	10,000
Read	31.56	94.95	193.04
Write	7.93	18.59	45.91
Pivot	56.62	159.55	319.34
Triangular solve	50.18	136.41	218.77
Matrix multiply	28.37	118.79	242.29
Factorization	22.74	45.18	63.87
Total	222.48	615.67	1158.39

Table 2: Timings in seconds for the main phases of out-of-core LU factorization of $M \times M$ matrices. Results are shown for $M = 5000$, 8000 and 10000. In all cases $n_b = 50$, $n_g = 2$, $P = 8$, and $Q = 8$. The version of the algorithm that stores the matrix in unpivoted form and performs pivoting on the fly was used. The out-of-core matrix was physically and logically distributed.

superblocks exceeds that required to hold the entire matrix, i.e., if

$$2.\frac{M}{P}.n_g.n_b < \frac{M}{P}.\frac{M}{Q},$$

or $2Qn_gn_b < M$. Thus, for the parameters of Table 4 the $M = 5000$ and $M = 8000$ cases fit in core, so we just read in the whole matrix, factorize it using the standard ScaLAPACK routine P_GETRF, and then write it out again. In Table 4 it takes about 58 seconds to perform an in-core factorization of a 5000×5000 matrix, compared with 191 seconds for an out-of-core factorization (see table 3). The $M = 8000$ case in Table 4 failed, presumably because PFS was not able to handle the need to simultaneously read 8 Mbytes from each of 64 separate files. The $M = 10000$ case ran successfully out-of-core, and the results in Table 4 should be compared with those in Table 3, from which we observe that increasing n_g increases the time for I/O and factorization, but decreases the times for all other phases of the algorithm. The increase in I/O is an unexpected result since increasing n_g should decrease the I/O cost. Perhaps the larger value of n_g increases the I/O cost because larger amounts of data are being read and written, leading to congestion in the parallel I/O system.

To understand the effect of varying the superblock width on the time for the triangular solve, matrix multiplication, and factorization phases of the algorithm we derive the following expressions for the number of floating-point operations in each phase,

$$\begin{array}{lcr} \text{Triangular solve:} & = & \frac{1}{2}M^2n_b - \frac{1}{2}Mn_b^2 \\ \text{Matrix multiply:} & = & \frac{2}{3}M^3 - M^2n_b + \frac{1}{3}Mn_b^2 \\ \text{Factorization:} & = & \frac{1}{2}M^2n_b + \frac{1}{6}Mn_b^2 \end{array}$$

These expressions apply in the sequential case ($Q = n_g = 1$), but the corresponding expression for the parallel algorihm is obtained by replacing n_b by Qn_bn_g. It should be noted that the total floating-point operation count for all three computational phases is $(2/3)M^3$, but the above expressions show that the way these operations are distributed among the phases depends on the width of the superblock, n_b. Thus, an increase in the superblock width results in an increase in the factorization time, and a decrease in the time for matrix multiplication. If the superblock width is sufficiently small compared with the matrix size then a small increase results in an increase

Task	5,000	8,000	10,000
Read	33.36	95.20	181.61
Write	18.85	53.87	117.91
Pivot	11.01	28.98	47.19
Triangular solve	50.20	136.65	218.74
Matrix multiply	28.38	118.55	242.21
Factorization	22.70	45.24	63.91
Total	191.46	549.94	977.05

Table 3: Timings in seconds for the main phases of out-of-core LU factorization of $M \times M$ matrices. Results are shown for $M = 5000$, 8000 and 10000. In all cases $n_b = 50$, $n_g = 2$, $P = 8$, and $Q = 8$. The version of the algorithm that stores the matrix in pivoted form was used. The out-of-core matrix was physically and logically distributed.

Task	5,000	8,000	10,000
Read	20.93	Fail	273.08
Write	59.39		238.66
Pivot	—		23.89
Triangular solve	—		177.48
Matrix multiply	—		117.24
Factorization	58.47		138.62
Total	148.86		1104.66

Table 4: Timings in seconds for the main phases of out-of-core LU factorization of $M \times M$ matrices. Results are shown for $M = 5000$, 8000 and 10000. In all cases $n_b = 50$, $n_g = 10$, $P = 8$, and $Q = 8$. The version of the algorithm that stores the matrix in pivoted form was used. Note that the $M = 5000$ and 8000 cases ran in-core, and that the $M = 8000$ case failed. The out-of-core matrix was physically and logically distributed.

in the triangular solve time. However, if the superblock width is large an increase will decrease the triangular solve time. It should be remembered that all three of these phases are running in parallel so communication time also influences the total running time. In general, increasing the n_b or n_g should decrease communication time on the Paragon as data are communicated in larger blocks. If the times for the computational phases in Tables 3 and 4 are summed we get about 524 seconds for $n_g = 2$ and about 432 seconds for $n_g = 10$ which suggests that a larger value of n_g results in more efficient parallel compputation overall. Communication overhead, together with the floating-point operation count, determines the performance of the computational phases of the algorithm as n_g changes.

The failure of the $M = 8000$ case in Table 3 prompted us to devise a second way of implementing logically distributed files. Instead of opening a separate file for each process, the new method opens a single file and divides it into blocks, assigning one block to each process. This does not change the user interface to the BLAPIOS described in Sec. 5. We refer to this type of file as a physically shared, logically distributed file. It should be noted that the terms "physically shared" and "physically distributed" refer to the view of the parallel file system from within the BLAPIOS. At the hardware level the file, or files, may be striped across multiple disks, as is the case for the Intel Paragon.

The rest of the results presented in this section are for physically shared, logically distributed files, and the version of the algorithm that stores the matrix in pivoted form. In Tables 5 and 6 results are presented for the same problems on 4×4 and 8×8 process meshes. It is interesting to note that increasing the number of processors from 16 to 64 results in only a very small decrease in the time for the triangular solve phase, indicating that the parallel efficiency for this phase is low. This is in contrast with the matrix multiplication phase which exhibits almost perfect speedup.

In Table 7 timings are presented for the case $n_g = 10$ for an 8×8 process mesh. Comparing these results first with those given in Table 4 for a physically and logically distributed file, the decrease in the times for reading and writing is striking. Secondly, of course, the physically shared case no longer fails for the $M = 8000$ in-core case. Comparison between Tables 6 and 7 shows that a for physically shared file an increase in n_g results in a decrease in I/O time, as expected from the dependency of the I/O time on M^3/n_b. However, the decrease is less than the expected factor of 5, particularly for

Task	5,000	8,000	10,000
Read	61.45	178.43	303.99
Write	36.61	124.11	211.67
Pivot	22.59	60.20	94.17
Triangular solve	52.84	139.09	219.66
Matrix multiply	114.70	482.79	948.93
Factorization	29.16	64.00	93.92
Total	350.12	1149.64	2042.41

Table 5: Timings in seconds for the main phases of out-of-core LU factorization of $M \times M$ matrices. Results are shown for $M = 5000$, 8000 and 10000. In all cases $n_b = 50$, $n_g = 2$, $P = 4$, and $Q = 4$. The version of the algorithm that stores the matrix in pivoted form was used. The out-of-core matrix was logically distributed, but physically shared.

Task	5,000	8,000	10,000
Read	34.29	95.74	201.18
Write	24.35	62.53	130.08
Pivot	10.94	28.85	47.27
Triangular solve	50.20	136.45	218.82
Matrix multiply	28.34	118.72	242.36
Factorization	22.70	45.05	63.87
Total	200.26	536.89	1006.34

Table 6: Timings in seconds for the main phases of out-of-core LU factorization of $M \times M$ matrices. Results are shown for $M = 5000$, 8000 and 10000. In all cases $n_b = 50$, $n_g = 2$, $P = 8$, and $Q = 8$. The version of the algorithm that stores the matrix in pivoted form was used. The out-of-core matrix was logically distributed, but physically shared.

the writes. Results in Table 8 for the case $n_g = 5$ show a read time for the $M = 10000$ case which is about the same as for $n_g = 10$, and a write time that is substantially less. This again shows that as n_g increases, thereby increasing the amount of data being read and written in each I/O operation, I/O performance starts to degrade quite significantly once n_g is sufficiently large.

Table 8 shows timings for the $M = 10000$ case for the same problem parameters as in Table 7, but for $n_g = 5$. Comparing the results in Tables 6, 7, and 8 we see that the time for writing data does not decrease montonically as n_g increase, but is smallest for $n_g = 5$. Again we ascribe this behavior to the apparent degradation in I/O performance when the volume of simultaneous I/O is large.

Task	5,000	8,000	10,000
Read	4.16	11.10	75.04
Write	3.59	14.25	99.60
Pivot	—	—	24.13
Triangular solve	—	—	180.25
Matrix multiply	—	—	130.12
Factorization	58.57	181.55	141.17
Total	69.47	206.90	709.22

Table 7: Timings in seconds for the main phases of out-of-core LU factorization of $M \times M$ matrices. Results are shown for $M = 5000$, 8000 and 10000. In all cases $n_b = 50$, $n_g = 10$, $P = 8$, and $Q = 8$. The version of the algorithm that stores the matrix in pivoted form was used. Note that the $M = 5000$ and 8000 cases ran in-core. The out-of-core matrix was logically distributed, but physically shared.

8 Summary and Conclusions

In this paper we have described a parallel left-looking algorithm for performing the out-of-core LU factorization of dense matrices. Use of out-of-core

Task	5,000	8,000	10,000
Read	—	—	77.92
Write	—	—	56.30
Pivot	—	—	32.51
Triangular solve	—	—	209.22
Matrix multiply	—	—	176.60
Factorization	—	—	92.69
Total	—	—	681.89

Table 8: Timings in seconds for the main phases of out-of-core LU factorization of $M \times M$ matrices. Results are shown for $M = 10000$ with $n_b = 50$, $n_g = 5$, $P = 8$, and $Q = 8$. The version of the algorithm that stores the matrix in pivoted form was used. The out-of-core matrix was logically distributed, but physically shared.

storage adds an extra layer to the hierarchical memory. In order to manage flexible and efficient access to this extra layer of memory an extra level of partitioning over matrix columns has been introduced into the standard ScaLAPACK algorithm. This is represented by the superblocks in the hybrid algorithm that we have described. The hybrid algorithm is left-looking at the outermost loop level, but uses a right-looking algorithm to factor the individual superblocks. This permits the trade-offs between I/O cost, communication cost, and load imbalance overhead to be controlled at the application level by varying the parameters of the data distribution and the superblock width.

We have implemented the out-of-core LU factorization algorithm on an Intel Paragon parallel computer. The implementation makes use of a small library of parallel I/O routines called the BLAPIOS, together with ScaLAPACK and PBLAS routines. From a preliminary performance study we have observed the following.

1. On the Paragon the version of the algorithm that stores the matrix in pivoted form is faster than the version that stores matrices in unpivoted form.

2. On the Paragon the parallel I/O system cannot efficiently and reliably

manage large numbers of open files if the volume of data being read is sufficiently large. We have therefore implemented logically distributed files using a single file partitioned among the processes.

3. We have a broad qualitative understanding of the performance. Increasing the superblock width by increasing n_g should decrease I/O costs, but this was found to be true only up to a point on the Paragon because when the volume of parallel I/O becomes too great, I/O performance starts to degrade. Thus, although it might be expected that the optimal approach would be a make the superblock as large as possible, this will not be fastest on all systems.

Future work will follow two main directions. We will seek to implement our out-of-core algorithm on other platforms, such as the IBM SP-2, symmetric multiprocessors, and clusters of workstations. The use of the MPI-IO library will be considered as a means of providing portability for our code, rather than implementing the BLAPIOS directly on each machine. We will also develop a more sophisticated analytical performance model, and use it to interpret our timings. The IBM SP-2 will be of particular interest as each processor is attached to its own disk. Hence, unlike our Paragon implementation, it may prove appropriate on the IBM SP-2 to implement logically distributed matrices as physically distributed matrices.

As network bandwidths continue to improve, networks of workstations may prove to be a good environment for research groups needing to perform very large LU factorizations. Such a system is cost-effective compared with supercomputers such as the Intel Paragon, and is under the immediate control of the researchers using it. Moreover, disk storage is cheap and easy to install. Consider the system requirements if we want to factor a $10^5 \times 10^5$ matrix in 24 hours. In a balanced system we might expect to spend 8 hours computing, 8 hours communicating over the network, and 8 hours doing I/O. Such a computation would require about 6.7×10^{14} floating-point operations, or 23 Gflop/s. If there are N_p workstations and each has 128 Mbytes of memory, then the maximum superblock width is $80N_p$ elements. The I/O per workstation is then,

$$8 \times \left(\frac{1}{2}\right)\left(\frac{M^3}{80N_p}\right)\left(\frac{1}{N_p}\right)$$

or $50000/N_p^2$ Gbyte per workstation. The total amount of data communicated between processes can be approximated by the communication volume of the matrix multiplication operations that asymptotically dominate. The total amount of communication is approximately $(2/3)(M^3/w_{sb})$ elements, where w_{sb} is the superblock width. Assuming again that the superblock width is $w_{sb} = 80N_p$, the total amount of communication is approximately $(1/120)(M^3/N_p)$ elements. So for 16 workstations, each would need to compute at about 1.5 Gflop/s, and perform I/O at about 6.8 Mbyte/s. A network bandwidth of about 145 Mbyte/s would be required. Each workstation would require 5 Gbyte of disk storage. These requirements are close to the capabilities of current workstation networks.

References

[1] E. Anderson, Z. Bai, C. H. Bischof, J. Demmel, J. J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. C. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, PA, USA, 2nd edition, 1995.

[2] J. Choi, J. Demmel, I. Dhillon, J. J. Dongarra, S. Ostrouchov, A. P. Petitet, K. Stanley, D. W. Walker, and R. C. Whaley. ScaLAPACK: a portable linear algebra library for distributed memory computers – design issues and performance. LAPACK Working Note No.95. Technical Report CS-95-283, Department of Computer Science, University of Tennessee, 107 Ayres Hall, Knoxville, TN 37996-1301, USA, 1995.

[3] J. Choi, J. J. Dongarra, S. Ostrouchov, A. P. Petitet, D. W. Walker, and R. C. Whaley. A proposal for a set of Parallel Basic Linear Algebra Subprograms. LAPACK Working Note No.100. Technical Report CS-95-292, Department of Computer Science, University of Tennessee, 107 Ayres Hall, Knoxville, TN 37996-1301, USA, 1995.

[4] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker. ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers. In *Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation*, pages 120–127. IEEE Computer Society Press, 1992.

[5] Peter Corbett, Dror Feitelson, Sam Fineberg, Yarsun Hsu, Bill Nitzberg, Jean-Pierre Prost, Marc Snir, Bernard Traversat, and Parkson Wong. Overview of the MPI-IO parallel I/O interface. In *IPPS '95 Workshop on Input/Output in Parallel and Distributed Systems*, pages 1–15, April 1995.

[6] J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Software*, 16:1–28, 1990. (Algorithm 679).

[7] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of FORTRAN Basic Linear Algebra Subprograms. *ACM Trans. Math. Software*, 14:1–32, 1988. (Algorithm 656).

[8] J. J. Dongarra, F. G. Gustavson, and A. Karp. Implementing linear algebra algorithms for dense matrices on a vector pipeline machine. *SIAM Review*, 26:91–112, 1984.

[9] J. J. Dongarra and R. C. Whaley. A users' guide to the BLACS v1.0. LAPACK Working Note No.94. Technical Report CS-95-281, Department of Computer Science, University of Tennessee, 107 Ayres Hall, Knoxville, TN 37996-1301, USA, 1995.

[10] A. Edelman. Large dense numerical linear algebra in 1993: The parallel computing influence. *International Journal Supercomputer Applications*, 7:113–128, 1993.

[11] A. Edelman. Large dense numerical linear algebra in 1994: The continuing influence of parallel computing. In *Proceedings of the 1994 Scalable High Performance Computing Conference*, pages 781–787. IEEE Computer Society Press, 1994.

[12] Proceedings of the Third Annual Workshop on I/O in Parallel and Distributed Systems. Held in conjunction with IPPS'95, Santa Barbara, April 1995.

[13] J. Ortega and C. Romine. The *ijk* forms of factorization II. Parallel systems. *Parallel Computing*, 7(2):149–162, 1988.

[14] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. J. Dongarra. *MPI: The Complete Reference.* MIT Press, Cambridge, MA, USA, 1996.

[15] S. Toledo and F. Gustavson. The design and implementation of SOLAR, a portable library for scalable out-of-core linear algebra computations. In *Fourth Annual Workshop on I/O in Parallel and Distributed Systems.* ACM Press, May 1996.