LOCALITY OF REFERENCE IN LU DECOMPOSITION WITH PARTIAL PIVOTING*

SIVAN TOLEDO†

Abstract. This paper presents a new partitioned algorithm for LU decomposition with partial pivoting. The new algorithm, called the recursively-partitioned algorithm, is based on a recursive partitioning of the matrix. The paper analyzes the locality of reference in the new algorithm and the locality of reference in a known and widely used partitioned algorithm for LU decomposition called the right-looking algorithm. The analysis reveals that the new algorithm performs a factor of $\Theta(\sqrt{M/n})$ fewer I/O operations (or cache misses) than the right-looking algorithm, where n is the order of the matrix and M is the size of primary memory. The analysis also determines the optimal block size for the right-looking algorithm. Experimental comparisons between the new algorithm and the right-looking algorithm show that an implementation of the new algorithm outperforms a similarly coded right-looking algorithm on 6 different RISC architectures, that the new algorithm performs fewer cache misses than any other algorithm tested, and that it benefits more from Strassen's matrix-multiplication algorithm.

Key words. LU factorization, Gaussian elimination, partial pivoting, locality of reference, cache misses

AMS subject classifications. 15A23, 65F05, 65Y10, 65Y20

1. Introduction.

Algorithms that partition dense matrices into blocks and operate on entire blocks as much as possible are key to obtaining high performance on computers with hierarchical memory systems. Partitioning a matrix into blocks creates temporal locality of reference in the algorithm and reduces the number of words that must be transferred between primary and secondary memories. This paper describes a new partitioned algorithm for LU-factorization with partial pivoting, called the **recursively-partitioned** algorithm. The paper also analyzes the number of data transfers in a popular partitioned LU-factorization algorithm, the so-called **right-looking** algorithm, which is used in LAPACK [1]. The performance characteristics of other popular partitioned LU-factorization algorithms, in particular Crout and the left-looking algorithm used in the NAG library [4], are similar to those of the right-looking algorithm so they are not analyzed.

The analysis of the two algorithms leads to two interesting conclusions. First, there is a simple system-independent formula for choosing the block size for the right-looking algorithm which is almost always optimal. Second, the recursively-partitioned algorithm generates asymptotically less memory traffic between memories than the right-looking algorithm, even if the block size for the right looking algorithm is chosen optimally. Numerical experiments indicate that the recursively-partitioned algorithm generates fewer cache misses and runs faster than the right-looking algorithm.

The recursively-partitioned algorithm computes the LU decomposition with partial pivoting of an n-by-m matrix while transferring only $\Theta(nm^2/\sqrt{M} + nm\lg m)$ words between primary and secondary memories, where M is the size of the primary memory. The right-looking algorithm, on the other hand, transfers at least $\Theta(\max(nm^2/\sqrt{M}, nm^{1.5}))$ words. The number of words actually transferred by conventional algorithms depends on a parameter r, which is not chosen optimally in

*Parts of this research were performed while the author was a postdoctoral fellow at the IBM T.J. Watson Research Center and a postdoctoral associate at the MIT Laboratory for Computer Science. The work at MIT was supported in part by ARPA under Grant N00014–94–1–0985.

†Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304.

LAPACK. The new algorithm is optimal in the sense that the number of words that it transfers is asymptotically the same as the number transferred by partitioned (or blocked) algorithms for matrix multiplication and solution of triangular systems (at least when the number of columns is not very small compared to the size of primary memory). The right looking algorithm achieves such performance only when the matrix is so large that a few rows fill the primary memory.

The recursively-partitioned algorithm algorithm has other advantages over conventional algorithms. It has no block-size parameter that must be tuned in order to achieve high performance. Since it is recursive, it is likely to perform better when the memory system has more than two levels, for example, on computer systems with two levels of cache or with both cache and virtual memory.

To understand the main idea behind the new algorithm, let us look first at the conventional right-looking LU factorization algorithm. The algorithm decomposes the input matrix into $\lceil n/r \rceil$ blocks of at most r columns. Starting from the leftmost block of columns, the algorithm iteratively factors a block of r columns using a column-oriented algorithm. After a block is factored, the algorithm updates the entire trailing submatrix. The parameter r must be carefully chosen to minimize the number of words transferred between memories. If r is larger than M/n, many words must be transferred when a block of columns is factored. If r is too small, many trailing submatrices must be updated, and most of the updates require the entire trailing submatrix to be read from secondary memory.

The main insight behind the recursively-partitioned algorithm is that there is no need to update the entire trailing submatrix after a block of columns is factored. After factoring the first column of the matrix, the algorithm updates just the next column to the right, which enables it to proceed. Once the second column is factored, we must apply the updates from the first two columns before we can proceed. The algorithm updates two more columns and proceeds. Once four columns are factored, they are used to update four more, and so on. In other words, the algorithm does not look all the way to the right every time a few columns are factored. As we shall see below, this short-sighted approach pays off.

From another point of view, the new algorithm is a recursive algorithm. We know that the larger r (the number of columns in a block), the smaller the number of data transfers required for updating trailing submatrices. The algorithm therefore chooses the largest possible size, $r = m/2$. If that many columns do not fit within primary memory, they are factored recursively using the same algorithm, rather than being factored using a naive column oriented algorithm. Once the left $m/2$ columns are factored, they are used to update the right $m/2$ columns which are subsequently factored.

The rest of the paper is organized as follows. Section 2 describes and analyzes the recursively-partitioned algorithm. Section 3 analyzes the block-column right-looking algorithm. The actual performance of LAPACK's right-looking algorithm and the performance of the recursively-partitioned algorithm are compared in Section 4 on several high-end workstations. Section 5 concludes the paper with a discussion of the results and of related research.

2. Recursively-Partitioned LU Factorization. The recursively-partitioned algorithm is not only more efficient than conventional partitioned algorithms, but it is also simpler to describe and analyze. This section first describes the algorithm, and then analyzes the complexity of the algorithm in terms of arithmetic operations and in terms of the amount of data transferred between memories during its execution.

The algorithm. The algorithm factors an n-by-m matrix A into an n-by-n permutation matrix P, an n-by-m unit lower triangular matrix L (that is, L's upper triangle is all zeros), and an m-by-m upper triangular matrix U, such that $PA = LU$. A is treated as a block matrix

$$A = \begin{bmatrix} A_{11} & A_{21} \\ A_{21} & A_{22} \end{bmatrix} ,$$

where A_{11} is a square matrix of order $m/2$-by-$m/2$.

1. If $m = 1$ then factor (that is, perform pivoting and scaling)

$$P_1 \begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix} = \begin{bmatrix} L_{11} \\ L_{21} \end{bmatrix} U_{11}$$

   and return.
2. Else, recursively factor

$$P_1 \begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix} = \begin{bmatrix} L_{11} \\ L_{21} \end{bmatrix} U_{11}$$

3. Permute

$$\begin{bmatrix} A'_{12} \\ A'_{22} \end{bmatrix} \leftarrow P_1 \begin{bmatrix} A_{12} \\ A_{22} \end{bmatrix} .$$

4. Solve the triangular system $L_{11}U_{12} = A'_{12}$ for U_{12}.
5. $A''_{22} \leftarrow A'_{22} - L_{21}U_{12}$.
6. Recursively factor $P_2 A''_{22} = L_{22}U_{22}$.
7. Permute $L'_{21} \leftarrow P_2 L_{21}$.
8. Return

$$P_2 P_1 \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L'_{21} & L_{22} \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{bmatrix} .$$

Complexity Analysis. It is not hard to see that the algorithm is numerically equivalent to the conventional column-oriented algorithm. Therefore, the algorithm has the name numerical properties as the conventional algorithm, and it performs same number of floating point operations, about $nm^2 - m^3/3$. In fact, all the variants of the LU-factorization algorithm discussed in this paper are essentially different schedules for the same algorithm. That is, they all have the same data-flow graph.

We now analyze the number of words that must be transferred between the primary and secondary memories for $n \geq m$. The size of primary memory is denoted by M. For ease of exposition, we assume that the number of columns is a power of two. We denote the number of words that the algorithm must transfer between memories by $\mathbf{IO}_{\mathbf{RP}}(n, m)$. We denote the number of words that must be transferred to solve an n-by-n triangular linear system with m right hand sides where the solution overwrites the right hand side by $\mathbf{IO}_{\mathbf{TS}}(n, m)$. We denote the number of words that must be transferred to multiply multiply an n-by-$-m$ matrix by an m-by-k matrix and add the result to an n-by-k matrix by $\mathbf{IO}_{\mathbf{MM}}(n, m, k)$.

Since the factorization algorithm uses matrix multiplication and solution of triangular linear system as subroutines, the number of I/O's it performs depends on the number of I/O's performed by these subroutines. A partitioned algorithm for solving triangular linear systems performs at most

$$\mathbf{IO_{TS}}(m,m) \le \begin{cases} 2.5m^2 & \text{if } m \le \sqrt{M/3} \\ \frac{m^3}{\sqrt{M/3}} + m^2 & \text{if } m \ge \sqrt{M/3} \end{cases} \tag{2.1}$$

I/O's. The actual number of I/O's performed is smaller, since the real crossover point is $\sqrt{M/2}$, not $\sqrt{M/3}$. Incorporating the improved bound into the analysis complicates the analysis with little effect on the final outcome. The number of I/O's performed by a standard matrix-multiplication algorithm is at most

$$\mathbf{IO_{MM}}(n,n,m) \le \begin{cases} 3nm + m^2 & \text{if } m \le \sqrt{M/3} \\ 2\frac{nm^2}{\sqrt{M/3}} + 2nm & \text{if } m \ge \sqrt{M/3} \end{cases} . \tag{2.2}$$

The bound for matrix multiplication holds for all values of $n \ge m$. The analysis here assumes the use of a conventional triangular solver and matrix multiplication, rather than so-called "fast" or Strassen-like algorithms. The asymptotic bounds for fast matrix-multiplication algorithms are better [5].

We analyze the recursively-partitioned algorithm using induction. Initially, the analysis that does not take into account the permutation of rows that the algorithm performs. We shall return to these permutations later in this section. The recurrence that governs the total number of words that are transferred by the algorithm is

$$\begin{aligned} \mathbf{IO_{RP}}(n,1) &= 2n \ , \\ \mathbf{IO_{RP}}(n,m) &= \mathbf{IO_{RP}}(n,m/2) + \mathbf{IO_{RP}}(n-m/2,m/2) \\ &\quad + \mathbf{IO_{TS}}(m/2,m/2)) \\ &\quad + \mathbf{IO_{MM}}(n-m/2,m/2,m/2)) \ . \end{aligned}$$

We first prove by induction that if $1/2 \le m/2 \le \sqrt{M/3}$, then $\mathbf{IO_{RP}}(n,m) \le 2nm(1+\lg m)$. The base case $m=1$ is true. Assuming that the claim is true for $m/2$, for $m \ge 2$ we have

$$\begin{aligned} \mathbf{IO_{RP}}(n,m) &\le 2n\frac{m}{2}\lg m + 2(n-m/2)\frac{m}{2}\lg m \\ &\quad + \frac{2.5m^2}{4} \\ &\quad + \frac{3nm}{2} - \frac{3m^2}{4} + \frac{m^2}{4} \\ &\le 2nm\lg m + \frac{3nm}{2} + (0.5 - 2\lg m)\frac{m^2}{4} \\ &\le 2nm(1+\lg m) \ . \end{aligned}$$

We now prove by induction that

$$\mathbf{IO_{RP}}(n,m) \le 2nm\left(\frac{m}{2\sqrt{M/3}} + \lg m\right)$$

for $m/2 \ge \sqrt{M/3}$. The claim is true for the base case $m/2 = \sqrt{M/3}$ since $m/2 \le \sqrt{M/3}$ and since $m/(2\sqrt{M/3}) = 1$. Assuming that the claim is true for $m/2$, we have

$$\mathbf{IO_{RP}}(n,m) \le 2nm\left(\frac{m}{4\sqrt{M/3}} + \lg m - 1\right)$$

$$
\begin{aligned}
&+\frac{m^3}{8\sqrt{M/3}}+\frac{m^2}{4} \\
&+\frac{2(n-m/2)m^2}{4\sqrt{M/3}}+\frac{2(n-m/2)m}{2} \\
&\le 2nm\left(\frac{m}{4\sqrt{M/3}}+\lg m-1\right) \\
&+\frac{m^3}{8\sqrt{M/3}}+\frac{m^2}{4} \\
&+\frac{nm^2}{2\sqrt{M/3}}-\frac{m^3}{4\sqrt{M/3}}+nm-\frac{m^2}{2} \\
&\le 2nm\left(\frac{m}{4\sqrt{M/3}}+\lg m-1\right) \\
&+\frac{nm^2}{2\sqrt{M/3}}-\frac{m^3}{8\sqrt{M/3}}+nm-\frac{m^2}{4} \\
&\le 2nm\left(\frac{m}{4\sqrt{M/3}}+\lg m\right) \\
&+\frac{nm^2}{2\sqrt{M/3}} \\
&= 2nm\left(\frac{m}{2\sqrt{M/3}}+\lg m\right) .
\end{aligned}
$$

To bound the number word transfers due to permutations we compute the number of permutations a column undergoes during the algorithm. Each column is permuted either in the factorization in Step 2 and in the permutation in Step 7, or in the permutation in Step 3 and in the factorization in Step 6. It follows that each column is permuted $1+\lg m$ times. If each word is brought from secondary memory, then the total number of I/O's required for permutations is at most $2n^2(1+\lg m)$. This bound can be achieved when $n < M$ by reading entire columns to primary memory and permuting them in primary memory.

The following theorem summarizes the main result of this section.

THEOREM 2.1. *Given a matrix multiplication subroutine whose I/O performance satisfies Equation (2.2) and a subroutine for solving triangular linear systems whose I/O performance satisfies Equation (2.1), the recursively-partitioned LU decomposition algorithm running on a computer with M words of primary memory computes the LU decomposition with partial pivoting of an n-by-m matrix using at most*

$$
\mathbf{IO}_{\mathbf{RP}}(n,m) \le 2nm\left(\frac{m}{2\sqrt{M/3}}+\lg m\right)+2n^2(1+\lg m)
$$

I/O's. □

3. Analysis of The Right-Looking LU Factorization. To put the performance of the recursively-partitioned algorithm in perspective, we now analyze the performance of the column-block right-looking algorithm. We first describe the algorithm and then analyze the number of data transfers, or I/O's, it performs. While the bounds we obtain are asymptotically tight, we focus on lower bounds in terms of the constants. The number of I/O's required during the solution of triangular linear systems is smaller than the number of I/O's required during the updates to the trailing submatrix (a rank r update to a matrix), so we ignore the triangular solves in the analysis.

Right-Looking LU. The algorithm factors an n-by-m matrix A such that $PA = LU$, where $n \geq m$. The algorithm factors r columns in every iteration. In the kth iteration we decompose A into

$$PA = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix},$$

where A_{11} is a square matrix of order $(k-1)r$ and A_{22} is a square matrix of order r. In the kth iteration the algorithm performs the following steps:

1. Factor
$$P_2 \begin{bmatrix} A_{22} \\ A_{32} \end{bmatrix} = \begin{bmatrix} L_{22} \\ L_{32} \end{bmatrix} U_{22} .$$
2. Permute
$$\begin{bmatrix} A_{23} \\ A_{33} \end{bmatrix} \leftarrow P_2 \begin{bmatrix} A_{23} \\ A_{33} \end{bmatrix} .$$
3. Permute
$$\begin{bmatrix} L_{21} \\ L_{31} \end{bmatrix} \leftarrow P_2 \begin{bmatrix} L_{21} \\ L_{31} \end{bmatrix} .$$
4. Solve the triangular system $L_{22}U_{23} = A_{23}$.
5. Update $A_{33} \leftarrow A_{33} - L_{32}U_{23}$.

The number of I/O's required to factor an n-by-r matrix using the column-by-column algorithm is

$$\frac{nr^2}{4} \leq \mathbf{IO}_{\mathbf{CF}}(n,r) \leq \frac{nr^2}{2}$$

when $M \leq nr/2$, but only

$$\mathbf{IO}_{\mathbf{CF}}(n,r) = 2nr$$

when $M \geq nr$. To simplify the analysis, we ignore the range of M in which more than half then matrix fits within primary memory but less then the entire matrix. (Using one level of recursion leads to $\Theta(nr)$ I/O's in this range). We use the facts that for $r \leq s$

$$\mathbf{IO}_{\mathbf{TS}}(r,s) = \begin{cases} 2rs + \frac{r^2}{2} & \text{if } r < \sqrt{M/3} \\ \frac{r^2 s}{\sqrt{M/3}} + rs & \text{if } r \geq \sqrt{M/3} \end{cases} \tag{3.1}$$

and that for $r \leq s \leq t$

$$\mathbf{IO}_{\mathbf{MM}}(t,r,s) = \begin{cases} 2ts + rs + rt & \text{if } r < \sqrt{M/3} \\ 2\frac{trs}{\sqrt{M/3}} + 2ts & \text{if } r \geq \sqrt{M/3} \end{cases} \tag{3.2}$$

The bound $2ts + rs + rt$ is an underestimate when $M < rs$. We ignore this small slack in the analysis.

The number of I/O's the algorithm performs depends on the relation of r to the dimensions of the matrix and to the size of memory. If r is so small that $M \geq nr$, then the updates to the trailing submatrix dominate the number of I/O's the algorithm performs. The $(m/r) - 1$ updates to the trailing submatrix require at least

$$\Theta(nm^2/r) = \Omega(n^2m^2/M)$$

I/O's. In particular, the first $m/2r$ updates require at least

$$\frac{m}{2r}2\left(n - \frac{m}{2}\right)\frac{m}{2} \geq \frac{nm}{M}\left(n - \frac{m}{2}\right)\frac{m}{2} = \frac{n^2m^2}{2M} - \frac{nm^3}{4M} \geq \frac{n^2m^2}{4M} \ .$$

If r is larger, factoring the m/r blocks of r columns requires at least

$$\frac{m}{r}\frac{nr^2}{4} = \frac{nmr}{4}$$

I/O's. The number of I/O's required for the rank-r updates depends on the value of r. If $M/n \leq r \leq \sqrt{M/3}$, then the total number of I/O's performed by the rank-r updates is at least

$$\frac{m}{2r}2\left(n - \frac{m}{2}\right)\frac{m}{2} \ .$$

Therefore, the number of I/O's performed by the algorithm is at least

$$\frac{nmr}{4} + \frac{m}{2r}2\left(n - \frac{m}{2}\right)\frac{m}{2} \ ,$$

which is minimized at

$$r_{\text{opt}} = \sqrt{2m - m^2/n} \ .$$

For $n \geq m$, the optimal value of r lies between

$$\sqrt{m} \leq r_{\text{opt}} \leq \sqrt{2m}$$

(The exact value might deviate slightly from this range, since the expression we derived for the number of I/O's is only a lower bound). Substituting the optimal value of r, we find that the algorithm performs at least

$$\left(\frac{1}{4} + \frac{1}{2\sqrt{2}}\right)nm^{1.5} - \frac{1}{4\sqrt{2}}m^{2.5} \geq \left(\frac{1}{4} + \frac{1}{4\sqrt{2}}\right)nm^{1.5}$$

I/O's in this range. If $\sqrt{m} < M/n$, then the value $r = M/n$ yields better performance than $\sqrt{m}$. If $\sqrt{m} > \sqrt{M/3}$, then the value $r = \sqrt{M/3}$ yields better performance than $\sqrt{m}$.

If r is yet larger, $r \geq \sqrt{M/3}$, then the rank-r updates require

$$\Theta((m/r)nmr/\sqrt{M/3}) = \Theta(nm^2/\sqrt{M/3})$$

I/O's. In particular, the first $m/2r$ updates require at least

$$\frac{m}{2r}\frac{2(n - m/2)(m/2)r}{\sqrt{M/3}} \geq \frac{nm^2}{2\sqrt{M/3}} - \frac{m^3}{4\sqrt{M/3}} \geq \frac{nm^2}{4\sqrt{M/3}}$$

I/O's. The total number of I/O's in this range, including both the updates and the factoring of blocks of columns, is therefore at least

$$\frac{nm^2}{4\sqrt{M/3}} + \frac{nmr}{4}$$

if $r \geq \sqrt{M/3}, M/n$. The number of I/O's is minimized by choosing the smallest possible r, $r_{\mathrm{opt}} = \sqrt{M/3}$.

If the matrix is not very large compared to the size of main memory, $n^2/3 \leq M$, it is also possible to choose r such that $\sqrt{M/3} \leq r \leq M/n$. In this case, the total number of I/O's is at least

$$\frac{nm^2}{4\sqrt{M/3}} + 2nm \geq \frac{n^2m^2}{4M} + 2nm \ .$$

The analysis can be summarized as follows. A value of r close to $\max(M/n, \sqrt{m})$ is optimal for almost all cases. The only exception is for truly huge matrices, where $M/3 \leq m$. For such matrices, $r = \sqrt{M/3}$ is better than $r = \sqrt{m}$. Combining the results, we obtain the following theorem.

THEOREM 3.1. *Given a matrix multiplication subroutine whose I/O performance satisfies Equation (3.1) and a subroutine for solving triangular linear systems whose I/O performance satisfies Equation (3.2), the right-looking LU decomposition algorithm running on a computer with M words of primary memory computes the LU decomposition with partial pivoting of an n-by-m matrix using at least*

$$\mathbf{IO_{RL}}(n,m) \geq \begin{cases} \frac{1}{4}\frac{n^2m^2}{M} & \text{if } r = M/n \\ \frac{1}{4}nm^{1.5} & \text{if } r \approx \sqrt{m} < \sqrt{M/3} \\ \frac{1}{4}nm^{1.5} & \text{if } r = \sqrt{M/3} \end{cases}$$

I/O's. □

The first case, $r = M/n$, leads to better performance only when more than $\sqrt{m}$ columns fit within primary memory. Although these are lower bounds, they are asymptotically tight. The value $1/4$ is a lower bound on the actual constant, which is higher than that.

4. Experimental Results. We have implemented and tested the recursively-partitioned algorithm[1]. The goal of the experiments was to determine whether the recursively partitioned algorithm is more efficient than the right-looking algorithm in practice. The results of the experiments clearly show that the recursively-partitioned algorithm performs less I/O and is that it is faster, at least on the computer on which the experiments were conducted.

The results of the experiments complement our analysis of the two algorithms. The analysis shows that the recursively-partitioned algorithm performs less I/O than the right looking algorithm for most values of n and M. The analysis stops short of demonstrating that one algorithm is faster than another in three respects. First, the bounds in the analysis are not exact. Second, the analysis counts the total number

[1] Our Fortran 90 implementation is available online by anonymous ftp from `theory.lcs.mit.edu` as `/pub/people/sivan/dgetrf90.f`. The code can be compiled by many Fortran 77 compilers, including compilers from IBM, Silicon Graphics, and Digital, by removing the RECURSIVE keyword and using a compiler option that enables recursion (see [11] for details).

of I/O's in the algorithm, but the distribution of the I/O within the algorithm is significant. Finally, the analysis uses a simplified model of a two-level hierarchical memory that does not capture all the subtleties of actual memory systems. The experiments show that even though our analysis is not exact in these respects, the recursively-partitioned algorithm is indeed faster.

Three sets of experiments are presented in this section. The first set presents and analyzes in detail experiments on IBM RS/6000 workstations. The goal of this set of experiments is to establish that the recursively-partitioned algorithm is faster than the right-looking algorithm. The second set of experiments show, in less detail, that the recursively-partitioned algorithm outperforms LAPACK's right-looking algorithm on a wide range of architectures. The goal of the second set of experiments is to establish the robustness of the performance of the recursively-partitioned algorithm. The third set of experiments shows that using Strassen's matrix multiplication algorithm speeds up the recursively-partitioned algorithm, but does not seem to speed up the right-looking algorithm.

Some of the technical details of the experiments, such as operating system versions, compiler versions, and compiler options are omitted from this paper. These details are fully described in our technical report [11].

Detailed Experimental Analyzes. The first set of experiments was performed on an IBM RS/6000 workstation with a 66.5 MHz POWER2 processor [14], 128 Kbytes 4-way set associative level-1 data-cache, a 1 Mbytes direct mapped level-2 cache, and a 128-bit-wide main memory bus. The POWER2 processor is capable of issuing two double-precision multiply-add instructions per clock cycle. Both LAPACK's right looking LU-factorization subroutine DGETRF and the recursively partitioned algorithm were compiled by IBM's XLF compiler version 3.2. All the algorithms used the BLAS from IBM's Engineering and Scientific Subroutine Library (ESSL). On square matrices we have also measured the performance of the LU-factorization subroutine DGEF from ESSL. The interface of this subroutine only allows for the factorization of square matrices. The coding style and the data structures used in the recursively-partitioned algorithm are the same as the ones used by LAPACK. In particular, permutations are represented in both algorithms as a sequence of exchanges. In all cases, the array that contains the matrix to be factored was allocated statically and aligned on a 16-byte boundary. The leading dimension of the matrix was equal to the number of rows (no padding).

The performance of the algorithms was assessed using measurements of both running time and cache misses. Time was measured using the machines real-time clock, which has a resolution of one cycle. The number of cache misses was measured using the POWER2 performance monitor [13]. The performance monitor is a hardware subsystem in the processor capable of counting cache misses and other processor events. Both the real-time clock and the performance monitor are oblivious to time sharing. To minimize the risk that measurements are influenced by other processes, we ran the experiments when no other users used the machine (but it was connected to the network). We later verified that the measurements are valid by comparing the real-time-clock measurements with the user time reported by AIX's `getrusage` system call on an experiment by experiment basis. All measurements reported here are based on an average of 10 executions.

We have coded two variants of the recursively-partitioned algorithm. The two versions differ in the way permutations are applied to submatrices. In one version, permutations are applied using LAPACK's auxiliary subroutine DLASWP. This sub-

TABLE 4.1

The performance in millions of operations per second (Mflops) and the number of cache misses per thousand floating point operations (CM/Kflop) of five LU-factorization algorithms on an IBM RS/6000 Workstation, on square matrices. The figures for LAPACK's DGETRF are those of the block size r with the best running time, in upright letters, and those of the block size with the smallest number of cache misses, in italics. The minimum number of cache misses does not generally coincide with the minimum running time. See the text for a full description of the experiments.

	$n = 1007$		$n = 1024$	
Subroutine	Mflops	CM/Kflop	Mflops	CM/Kflop
LAPACK's DGETRF, row exchanges	178, *176*	5.81, *5.65*	170, *168*	5.45, *5.29*
Recursively-partitioned, row exchanges	201	3.76	186	4.14
LAPACK's DGETRF, permuting by columns	201, *199*	2.94, *2.81*	198, *195*	3.11, *3.02*
Recursively-partitioned, permuting by columns	222	1.61	223	1.59
ESSL's DGEF	228	2.15	221	3.42

routine, which is also used by LAPACK's right-looking algorithm, permutes the rows of a submatrix by exchanging rows using the vector exchange subroutine DSWAP, a level-1 BLAS. The second version permutes the rows of the matrix by applying the entire sequence of exchanges to one column after another. The difference amounts to swapping the inner and outer loops. This change was suggested by Fred Gustavson.

The first experiment, whose results are summarized in Table 4.1, was designed to determine the effects of a complex hierarchical memory system on the partitioned algorithms. Four facts emerge from the table.

1. The recursively partitioned algorithm performs less cache misses and delivers higher performance than the right-looking algorithm. ESSL's subroutine performs less cache misses than LAPACK but more than the recursively-partitioned algorithm, but it achieves best or close to best performance.
2. Permuting one column at a time leads to less cache misses and faster execution than exchanging rows. This is true for both the right-looking algorithm and the recursively-partitioned algorithm. This is probably a result of the advantage of the stride-1 access to the column in the column permuting over the large stride access to rows in the row exchanges.
3. The performance in term of both time and cache misses of all the algorithms except the recursively-partitioned with column permuting is worse when the leading dimension of the matrix is a power of 2 than when it is not. The performance of the recursively-partitioned algorithm with column permuting improves by less than half a percent. The degradation in performance on a power of 2 is probably caused by fact that the caches are not fully associative.
4. The running time depends on the measured number of cache misses, but not completely. This can be seen both from the fact that ESSL's DGEF performs more cache misses than the recursively partitioned algorithm, but it is faster, and from the fact that the block size that leads to the minimum number of cache misses in the DGETRF does not lead to the best running time. The discrepancy can be caused by several factors that are not measured, including misses and conflicts in the level-2 cache, TLB misses, and instruction scheduling. In all four cases in the table the minimum running time is achieved with a value of r that is higher than the number that leads to a minimum number of cache misses. For example, on $n = 1007$, DGETRF with row exchanges performed the least number of cache misses with $r = 40$, but the fastest running time was achieved with $r = 55$. This may mean that

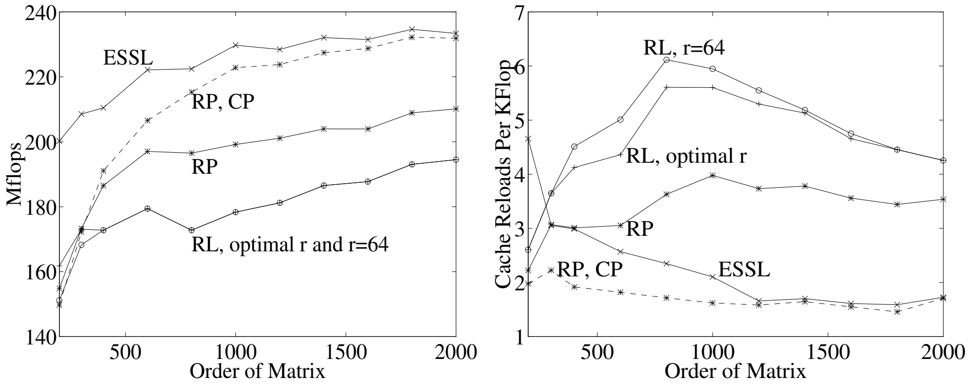

FIG. 4.1. *The performance in Mflops (on the left) and the number of cache misses per Kflop (on the right) of LU factorization algorithms on an IBM RS/6000 Workstation. These graphs depict the performance of the recursively-partitioned (PR) and right-looking (RL) algorithms on square matrices. The optimal value of r was selected experimentally from powers of 2 between 2 and 256. The dashed lines represent the performance of the recursively-partitioned algorithms with column permuting (CP).*

the cause of the discrepancy is misses in the level-2 cache, which is larger than the level-1 cache and therefore may favor a larger block size (since more columns fit in it).

In summary, the experiment shows that although the implementation details of the memory system influence the performance of the algorithms, the recursively-partitioned algorithm still emerges as faster than the right-looking one when they are implemented in a similar way.

The second set of experiments was designed to assess the performance of the algorithms over a wide range of input sizes. The performance and number of cache misses of the algorithms are presented in Figure 4.1 on square matrices ranging in order from 200 to 2000. The level-1 cache is large enough to store a matrix of order 128. The following points emerge from the experiment.

1. Beginning with matrices of order $n = 300$, the the recursively-partitioned algorithm with column permuting is faster than the same algorithm with row exchanges which is still faster than LAPACK's DGETRF with row exchanges (we did not measure the performance of DGETRF with column permuting in this experiment).
2. The performance of DGETRF with optimal block size r and with $r = 64$ is essentially the same except at $n = 300$, although the optimal block size clearly leads to a smaller number of cache misses from $n = 400$ through $n = 1600$.
3. The recursively-partitioned algorithm performs less cache misses than ESSL's DGEF on all input sizes, but it is not faster. As in the first experiment, the experiment itself does not indicate what causes this phenomenon. We speculate that it is caused by better instruction scheduling or fewer misses in the level-2 cache.

The next experiment was designed to determine the sensitivity of the performance of the right-looking algorithm to the block size r. We used the column permuting strategy which proved more efficient in the previous experiments. The experiment consists of running the algorithm on a range of block sizes on a square matrix of order 1007 and on a rectangular 62500-by-64 matrix. The factorization of a rectangular

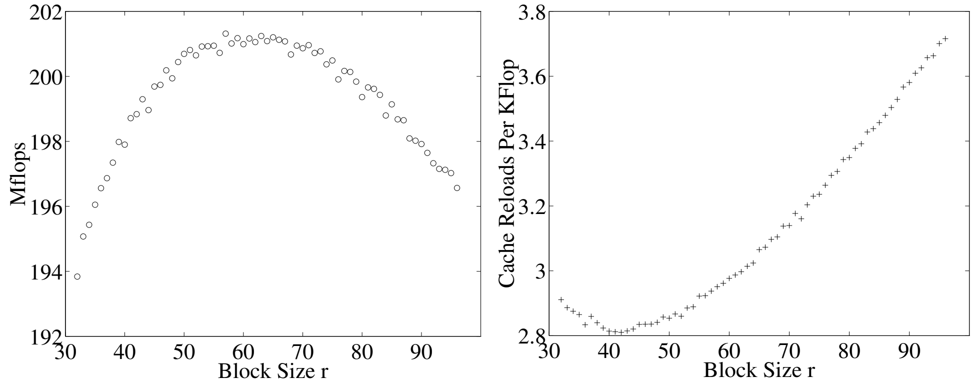

FIG. 4.2. *The performance in Mflops (on the left) and the number of cache misses per Kflop (on the right) of the right-looking algorithm with column permuting with as a function of the block size r. The order of the square matrix used is $n = 1007$. Note that the y-axes do not start from zero.*

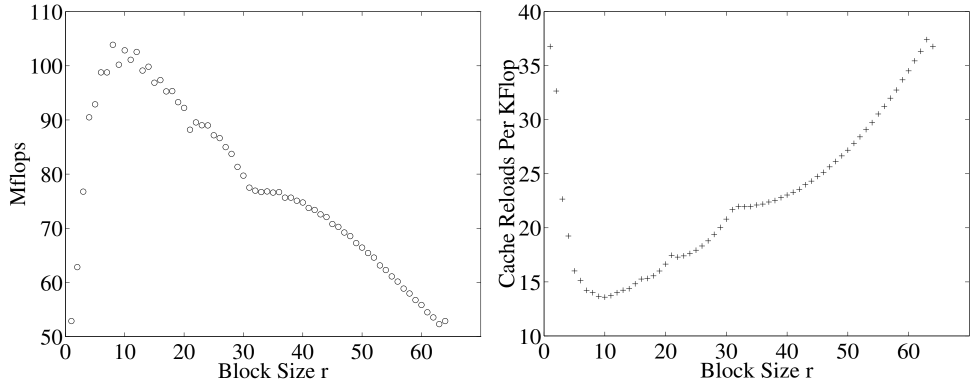

FIG. 4.3. *The performance in Mflops (on the left) and the number of cache misses per Kflop (on the right) of the right-looking algorithm with column permuting with as a function of the block size r. The dimensions of the matrix are 62500 by 64. For comparison, the performance of the recursively partitioned algorithm on this problem is 118 Mflops and 11.03 CM/Kflop.*

matrix with $n > m$ arises as a subproblem in out-of-core LU factorization algorithms that factor blocks of columns that fit within core. The specific dimensions of the matrices were chosen so as to minimize the effects of conflicts in the memory system on the results. The results for $n = m = 1007$, shown in Figures 4.2 show that the minimum number of cache misses occurs at $r = 42$, which is higher than $\sqrt{m} \approx 32$, and that the best performance is achieved with an even higher value of r, 55. The performance is not very sensitive to the choice of r, however, and all values between about 50 and 70 yield essentially the same performance, 201 Mflops. The results for 62500-by-64 matrices, shown in Figures 4.3, show that the minimum number of cache misses occur at $r = 10$, and the best performance occurs at $r = 8$, which happens to coincide exactly with $\sqrt{m}$. The sensitivity to r is greater here than in the square case, especially below the optimal value.

The last experiment in this set, presented in Figure 4.4, was designed to determine whether the discrepancy between the optimal block size in terms of level-1 cache misses

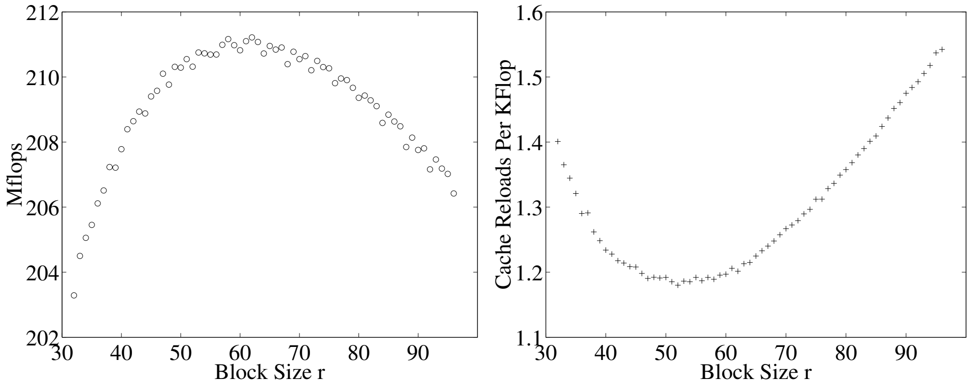

FIG. 4.4. *The performance in Mflops (on the left) and the number of cache misses per Kflop (on the right) of the right-looking algorithm with column permuting with as a function of the block size r. The order of the square matrix used is $n = 1007$. The machine used here has a bigger level-1 cache and no level-2 cache than the machine used in all the other experiments. Compare to Figure 4.2. For comparison, the performance of the recursively-partitioned algorithm on this problem on this machine is 229 Mflops and 0.650 CM/Kflop.*

and the optimal block size in terms of running time was caused by the level-2 cache. The experiment repeats the last experiment for square matrices of order 1007, except that the experiment was conducted on a machine with a 256-bit-wide main memory bus, 256 Kbytes level-1 cache, and no level-2 cache. The two machines are identical in all other respects. There is a discrepancy in optimal block sizes in Figure 4.4, but it is smaller than the discrepancy in Figure 4.2. The experiment shows that the discrepancy is *not* caused solely by the level-2 cache. It is not possible to determine whether the smaller discrepancy in this experiment is due to the lack of level-2 cache or to the larger level-1 cache.

Robustness Experiments. The second set of experiments show that the performance advantage of the recursively partitioned algorithm, which was demonstrated by the first set of experiments, is not limited to a single computer architecture. The experiments accomplish this goal by showing that the recursively partitioned algorithm outperforms the right looking algorithm on a wide range of architectures.

All the experiments in this set compare the performance of the recursively partitioned algorithm with the performance of LAPACK's right-looking on two sizes of square matrices, $n = 1007$ and $n = 2014$ (except when the larger matrices do not fit within main memory). These sizes were chosen so as to minimize the impact of cache associativity on the results. Each measurement reported represents the average of the best 5 out of 10 runs, to minimize the effect of other processes in the system. The block size for the right-looking algorithm was LAPACK's default $r = 64$.

We used the following machine configurations:

- A 66.5 MHz IBM RS/6000 workstation with a POWER2 processor, 128 Kbytes 4-way set associative data-cache, a 1 Mbytes direct mapped level-2 cache. and a 128-bit-wide bus. We used the BLAS from IBM's ESSL.
- A 25 MHz IBM RS/6000 workstation with a POWER processor, 64 Kbytes 4-way set associative data-cache, and a 128-bit-wide bus. We used the BLAS from IBM's ESSL.
- A 100 MHz Silicon Graphics Indy workstation with a MIPS R4600/R4610

TABLE 4.2

The running time in seconds of LU factorization algorithms on several machines. For each machine and each matrix order, the table shows the running times of the recursively-partitioned (RP) algorithm and the right-looking (RL) algorithm with row exchanges and column permutations. Some measurements are not available and marked as N/A because the amount of main memory is insufficient to factor the larger matrix in core. See the text for a full description of the experiments.

	$n = 1007$				$n = 2014$			
	Row Exchanges		Column Pivoting		Row Exchanges		Column Pivoting	
Machine	RL	RP	RL	RP	RL	RP	RL	RP
IBM POWER2	3.82	3.39	3.37	3.05	27.88	26.00	25.28	23.45
IBM POWER	22.81	19.07	17.87	16.86	146.1	143.4	135.8	129.7
SGI R4600/R4610	37.15	34.39	36.42	33.57	N/A	N/A	N/A	N/A
SGI R4400/R4010	9.44	8.36	9.38	8.29	73.92	68.64	73.73	66.79
DEC A21064	9.27	8.94	9.36	8.89	N/A	N/A	N/A	N/A
DEC A21164	2.61	2.56	2.30	2.25	20.15	19.68	17.80	17.06

CPU/FPU pair, a 16 Kbytes direct mapped data cache, and a 64-bit-wide bus. We used the SGI BLAS. This machine has only 32 Mbytes of main memory, so the experiment does not include matrices of order $n = 2014$.

- A 250 MHz Silicon Graphics Onyx workstation with 4 MIPS R4400/R4010 CPU/FPU pairs, a 16 Kbytes direct mapped data cache per processor, a 4 Mbytes level-2 cache per processor, and a 2-way interleaved main memory system with a 256-bit-wide bus. The experiment used only one processor. We used the SGI BLAS.
- A 150 MHz DEC 3000 Model 500 with an Alpha 21064 processor, 8 Kbytes direct mapped cache, and a 512 Kbytes level-2 cache. We used the BLAS from DEC's DXML for IEEE floating point. A limit on the amount of physical memory allocated to a process prevented us from running the experiment on matrices of order $n = 2014$.
- A 300 MHz Digital AlphaServer with 4 Alpha 21164 processors, each with an 8 Kbytes level-1 data cache, a 96 Kbytes on-chip level-2 cache, and a 4 Mbytes level-2 cache. The experiment used only one processor. We used the BLAS from DEC's DXML for IEEE floating point.

The results, which are reported in Table 4.2, show that the recursively partitioned algorithm consistently outperforms the right-looking algorithm. The results also show that permuting columns is almost always faster than exchanging rows.

Experiments using Strassen's Algorithm. Performing the updates of the trailing submatrix using a variant of Strassen's algorithm [10] improved the performance of the recursively partitioned algorithm. We replaced the call to DGEMM, the level-3 BLA subroutine for matrix multiply-add by a call to DGEMMB, a public domain implementation[2] of a variant of Strassen algorithm [3]. (Replacing the calls to DGEMM by calls to a Strassen matrix-multiplication subroutine in IBM's ESSL gave similar results). DGEMMB uses Strassen's algorithm only when all the dimensions of the input matrices are greater than a machine-dependent constant. The authors of DGEMMB set this constant to 192 for IBM RS/6000 workstations.

In the recursively-partitioned algorithm with column permuting, the replacement

[2]Available online from `http://www.netlib.org/linalg/gemmw`.

of DGEMM by DGEMMB reduced the factorization time on the POWER2 machine to 2.99 seconds for $n = 1007$ and to 22.18 seconds for $n = 2014$. The factorization times with the conventional matrix multiplication algorithm, reported in the first line of Table 4.2, are 3.05 and 23.45 seconds. The running time was reduced from 182.7 to 166.8 seconds on a matrix of order $n = 4028$. The change would have no effect on the right-looking algorithm, since in all the matrices it multiplies at least one dimension is r which was smaller than 192 in all the experiments.

A similar experiment carried out by Bailey, Lee, and Simon [2] showed that Strassen's algorithm can accelerate the LAPACK's right-looking LU factorization on a Cray Y-MP. The largest improvements in performance, however, occured when large values of r were used. The fastest factorization of a matrix of order $n = 2048$, for example, was obtained with $r = 512$. Such a value is likely to cause poor performance on machines with caches. (The Cray Y-MP has no cache.) On the IBM POWER2 machine, which has caches, increasing r from 64 to 512 causes the factorization time with a conventional matrix multiplication algorithm to increase from 30.8 seconds 54 seconds. Replacing the matrix multiplication subroutine by DGEMMB with $r = 512$ reduces the solution time, but by less than 2 seconds.

5. Conclusions. The recursively-partitioned algorithm should be used instead of the right-looking algorithm because it delivers similar or better performance without parameters that must be tuned. No parameter to choose means that there is no possibility of a poor choice, and hence the new algorithm is more robust. Section 4 shows that the performance of the right-looking algorithm can be sensitive to r, and that the best performance does not always coincide with the block size that causes the smallest number of cache misses. Choosing r can be especially difficult on machines with more than two levels of memory. A recursive algorithm, on the other hand, is a natural choice for hierarchical memory systems with more than two levels.

The recursively partitioned algorithm provides a good opportunity to use a fast matrix multiplication algorithm, such as Strassen's algorithm. Since a significant fraction of the work performed by the recursively partitioned algorithm is used to multiply large matrices, the benefit of using Strassen's algorithm can be large. The right-looking algorithm performs the same work by several multiplications of smaller matrices, so the benefit of Strassen's algorithm should be smaller.

The analysis of the right-looking algorithm in Section 3 shows how the block size r should be chosen. The value $r \approx \sqrt{m}$ is optimal with two exceptions. When a single row is too large to fit within primary memory, a value $r = \sqrt{M/3}$ leads to better performance. When more than $\sqrt{m}$ columns fit within primary memory, r should be set to M/n to minimize memory traffic. The extreme cases are the source of the difficulty in choosing a good value of r for hierarchical memory systems with more than two levels. In our experiments, the performance of the right-looking algorithm on matrices with more rows than columns was very sensitive to the choice of r, but it was not sensitive on large square matrices.

In the typical cases, when at least one row fits within primary memory, the right-looking algorithm with an optimal choice of r performs a factor of $\Theta(\sqrt{M/m})$ more data transfers than the recursively partitioned algorithm. In our experiments this factor led to a significant difference in both the number of cache misses and the running time.

The conclusion that the value $r = \sqrt{m}$ is often close to optimal shows that there is a system-independent way to choose r. In comparison, the model implementation of ILAENV, LAPACK's block-size-selection subroutine, uses a fixed value, 64, and LAPACK's *User's Guide* advises that system-dependent tuning of r could improve performance. The viewpoint of the LAPACK designers seems to be that r is a system-dependent parameter whose role is to hide the low bandwidth of the secondary memory system during the updates of the trailing submatrices. Our analysis here shows that the true role of r is to balance the number of data transfers between the two components of the algorithm: the factorization of blocks of columns and the updates of the trailing submatrices.

Designers of out-of-core LU decomposition codes often propose to use block-column (or row) algorithms. Many of them propose to choose $r = M/n$ so that an entire block of columns fits within primary memory [4, 6, 7, 15]. This approach works well when the columns are short and a large number of them fits within primary memory, but the performance of such algorithms would be unacceptable when only few columns fit within primary memory. Some researchers [7, 8, 9] suggest that algorithms that use less primary memory than is necessary for storing a few columns might have difficulty implementing partial pivoting. The analysis in this paper shows that it is possible to achieve a low number of data transfers even when a single row or column does not fit within primary memory.

Womble et al. [15] presented a recursively-partitioned LU decomposition algorithm without pivoting. They claimed, without a proof, that pivoting can be incorporated into the algorithm without asymptotically increasing the number of I/O's the algorithm performs. They suggested that a recursive algorithm would be difficult to implement, so they implemented instead a partitioned left-looking algorithm using $r = M/n$.

Toledo and Gustavson [12] describe a recursively-partitioned algorithm for out-of-core LU decomposition with partial pivoting. Their algorithm uses recursion on large submatrices, but switches to a left-looking variant on smaller submatrices (that would still not fit within main memory). Depending on the size of main memory, their algorithm can factor a matrix in 2/3 the amount of time used by an out-of-core left-looking algorithm with a fixed block size.

6. Acknowledgments. Thanks to Rob Schreiber for reading several early versions of this paper and commenting on them. Thanks to Fred Gustavson and Ramesh Agarwal for helpful suggestions. Thanks to the anonymous referees for several helpful comments.

REFERENCES

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, *LAPACK User's Guide*, SIAM, Philadelphia, PA, 2nd ed., 1994. Also available online from `http://www.netlib.org`.

[2] D. H. Bailey, K. Lee, and H. D. Simon, *Using Strassen's algorithm to accelerate the solution of linear systems*, J. of Supercomputing, 4 (1990), pp. 357–371.

[3] C. C. Douglas, M. Heroux, G. Slishman, and R. M. Smith, *GEMMW: A portable level 3 BLAS Winograd variant of Strassen's matrix-matrix multiply algorithm*, J. of Computational Physics, 110 (1994), pp. 1–10.

[4] J. J. Du Cruz, S. M. Nugent, J. K. Reid, and D. B. Taylor, *Solving large full sets of linear equations in a paged virtual store*, ACM Transactions on Mathematical Software, 7 (1981), pp. 527–536.

[5] P. C. Fischer and R. L. Probert, *A note on matrix multiplication in a paging environment*, in ACM '76: Proceedings of the Annual Conference, 1976, pp. 17–21.

[6] N. Geers and R. Klees, *Out-of-core solver for large dense nonsymmetric linear systems*, Manuscripta Geodetica, 18 (1993), pp. 331–342.

[7] R. G. Grimes, *Solving systems of large dense linear equations*, J. of Supercomputing, 1 (1988), pp. 291–299.

[8] A. C. McKeller and E. G. Coffman, Jr., *Organizing matrices and matrix operations for paged memory systems*, Communications of the ACM, 12 (1969), pp. 153–165.

[9] C. B. Moler, *Matrix computations with Fortran and paging*, Communications of the ACM, 15 (1972), pp. 268–270.

[10] V. Strassen, *Gaussian elimination is not optimal*, Numer. Math., 13 (1969), pp. 354–355.

[11] S. Toledo, *Locality of reference in LU decomposition with partial pivoting*, Tech. Report RC20344, IBM T.J. Watson Research Center, Yorktown Heights, NY, Jan. 1996.

[12] S. Toledo and F. G. Gustavson, *The design and implementation of SOLAR, a portable library for scalable out-of-core linear algebra computations*, in Proceedings of the 4th Annual Workshop on I/O in Parallel and Distributed Systems, Philadelphia, May 1996, pp. 28–40.

[13] E. H. Welbon, C. C. Chan-Nui, D. J. Shippy, and D. A. Hicks, *The POWER2 performance monitor*, IBM Journal of Research and Development, 38 (1994).

[14] S. W. White and S. Dhawan, *POWER2: Next generation of the RISC System/6000 family*, IBM Journal of Research and Development, 38 (1994).

[15] D. Womble, D. Greenberg, S. Wheat, and R. Riesen, *Beyond core: Making parallel computer I/O practical*, in Proceeings of the 1993 DAGS/PC Symposium, Hanover, NH, June 1993, Dartmouth Institute for Advanced Graduate Studies, pp. 56–63. Also available online from `http://www.cs.sandia.gov/~dewombl/parallel_io_dags93.html`.