Design and Scheduling of an Algorithm-by-Blocks for the LU Factorization on Multithreaded Architectures

FLAME Working Note #26

Gregorio Quintana-Ortí* Enrique S. Quintana-Ortí* Ernie Chan†
Robert A. van de Geijn† Field G. Van Zee †

September 19, 2007

Abstract

The scalable parallel implementation, targeting SMP and/or multicore architectures, of the LU factorization of a matrix is studied. It is shown that an algorithm-by-blocks exposes a higher degree of parallelism than traditional implementations based on multithreaded BLAS. This algorithm requires a different pivoting strategy, incremental pivoting, but allows most computation to be cast in terms of matrix-matrix multiplication without adversely increasing the operation count. Its implementation using the SuperMatrix runtime system is discussed and the scalability of the solution is demonstrated on an ccNUMA platform with 16 processors.

1 Introduction

With the emergence of parallel computing architectures with many processing elements (e.g., SMP systems with many processors, multicore chips with many cores, and CPUs featuring hardware accelerators such as the Cell processor [25, 2]), it is now widely recognized that commonly used libraries like LAPACK will need to be reimplemented, possibly from scratch. In this paper, we explore algorithmic modifications to the LU factorization with pivoting that support an algorithm-by-blocks. It is shown that this algorithm-by-blocks exhibits a high degree of parallelism that can be exploited by multithreaded architectures. This adds to a body of work that provides insights into how linear algebra algorithms in general can be rewritten to better utilize the compute power of systems with many processing cores [7, 30, 8, 31, 6, 5].

Traditional algorithms for the LU factorization with partial pivoting exhibit the property that, periodically, an updated column is required for a critical computation; in order to compute which row to pivot during the k-th iteration, the k-th column must have been updated with respect to all previous computation. This greatly restricts the order in which the computations can be performed.

The problem is compounded by the observation that, for scalability and data locality, it is beneficial to view, store, and compute with the matrix as a two dimensional array of submatrices (blocks) [7, 9, 30, 8, 6, 5]. Thus, the column needed for computing which row to pivot, as well as the row to be pivoted, likely span multiple blocks. This need for viewing and/or storing matrices by blocks was also observed for out-of-core dense linear algebra computations [35] and the implementation of dense linear algebra operations on distributed-memory architectures [38, 11].

The challenge we confront in this paper is that of developing a high performance LU factorization algorithm with pivoting while keeping the implementation simple. The contributions of this paper include:

*Departamento de Ingeniería y Ciencia de Computadores, Universidad Jaume I, 12.071 - Castellón (Spain), {gquintan,quintana}@icc.uji.es.

†Department of Computer Sciences, The University of Texas at Austin, Austin, Texas 78712, {echan,field,rvdg}@cs.utexas.edu.

- A demonstration that the LU factorization can attain high performance even when coded at a high level of abstraction and even when targeting complex environments such as manythreaded architectures.
- A study that compares and contrasts traditional blocked algorithms for the LU factorization, which extract parallelism within the Basic Linear Algebra Subprograms (BLAS) [26, 14, 13], to the pure algorithm-by-blocks first proposed in [23]. This algorithm is similar to the algorithm-by-blocks for the QR factorization proposed in [18], for which multithreaded parallel implementations are given in [30, 6].
- Further examples of how (a) the FLASH extension of FLAME/C supports storage by blocks for both types of algorithms and (b) the SuperMatrix runtime system supports, transparent to the programmer, out-of-order computation on blocks.
- A discussion on the numerical stability of the algorithm.
- An analysis that the extra work associated with the algorithms-by-blocks represents a lower order cost term, in contrast to a claim in [6].
- Performance that rivals that of an algorithms-by-blocks for the QR factorization, in contrast to performance reported in [5].

The remainder of the paper is structured as follows: Sections 2 and 3—taken essentially verbatim from [29]—review algorithms for the LU factorization with partial pivoting and how to modify them to factor a 2×2 matrix of blocks, using *incremental pivoting*, in order to update an existing LU factorization. Section 4 explains how updating an LU factorization can be extended to yield algorithms-by-blocks. The empirical data in Section 5 shows how incremental pivoting affects element growth and more generally the stability of the algorithm. Section 6 provides an overview of various tools and methods derived from the FLAME project which were used in the implementation (and also discussed in [30]). Performance results can be found in Section 7 and concluding remarks follow in the final section.

We adopt the following conventions: matrices, vectors, and scalars are denoted by upper-case, lower-case, and lower-case Greek letters, respectively. Algorithms are presented in a notation that we have developed as part of the FLAME project [17, 3]. If one keeps in mind that the thick lines in the partitioned matrices and vectors indicate how far the computation has proceeded, we believe the notation to be mostly intuitive. Otherwise, we suggest that the reader consult some of these related papers.

2 The LU Factorization with Partial Pivoting

Consider an $m \times n$ matrix A and its LU factorization with partial pivoting given by

$$PA = LU, \tag{1}$$

where P is a permutation matrix, L is lower trapezoidal, and U is upper triangular. The LU factorization is obtained by means of a triangularization procedure also known as Gaussian elimination [16, 33]. Here, a sequence of permutation matrices $P_1, P_2, \ldots, P_n$ and Gauss elimination matrices $L_1, L_2, \ldots, L_n$ are computed to reduce matrix A to upper triangular form. In practice, the factors L and U overwrite matrix A and the pivots are stored in an array of $\min(m, n)$ elements.

The LINPACK [12] implementation of (1) corresponds to an expression of the form

$$L_n^{-1} P_n \cdots L_2^{-1} P_2 L_1^{-1} P_1 A = U,$$

that does not provide the factor L explicitly. The LINPACK algorithm applies row permutations to A as the matrix is factorized. The LAPACK implementation provides the lower triangular factor by rearranging the computations in this expression as

$$\hat{L}_n^{-1} \cdots \hat{L}_2^{-1} \hat{L}_1^{-1} P_n \cdots P_2 P_1 A = L^{-1} PA = U.$$

Algorithm: $[A,\ p] := [\{L\backslash U\},\ p] = \mathrm{LU}_{\mathrm{UNB}}(A)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$ and $p \rightarrow \left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right)$

where A_{TL} is 0×0 and p_T has 0 elements

while $n(A_{TL}) < n(A)$ **do**

Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right) \text{ and } \left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right) \rightarrow \left(\begin{array}{c} p_0 \\ \hline \pi_1 \\ \hline p_2 \end{array}\right)$$

where α_{11} and π_1 are scalars

LINPACK variant:

$\left[\left(\begin{array}{c} \alpha_{11} \\ \hline a_{21} \end{array}\right), \pi_1\right] := \textsc{Pivot}\left(\begin{array}{c} \alpha_{11} \\ \hline a_{21} \end{array}\right)$

if $\alpha_{11} \neq 0$ **then**

$\left(\begin{array}{c} a_{12}^T \\ \hline A_{22} \end{array}\right) := P(\pi_1)\left(\begin{array}{c} a_{12}^T \\ \hline A_{22} \end{array}\right)$

$a_{21} := a_{21}/\alpha_{11}$

$A_{22} := A_{22} - a_{21}a_{12}^T$

endif

LAPACK variant:

$\left[\left(\begin{array}{c} \alpha_{11} \\ \hline a_{21} \end{array}\right), \pi_1\right] := \textsc{Pivot}\left(\begin{array}{c} \alpha_{11} \\ \hline a_{21} \end{array}\right)$

if $\alpha_{11} \neq 0$ **then**

$\left(\begin{array}{c|c} a_{10}^T & a_{12}^T \\ \hline A_{20} & A_{22} \end{array}\right) := P(\pi_1)\left(\begin{array}{c|c} a_{10}^T & a_{12}^T \\ \hline A_{20} & A_{22} \end{array}\right)$

$a_{21} := a_{21}/\alpha_{11}$

$A_{22} := A_{22} - a_{21}a_{12}^T$

endif

Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right) \text{ and } \left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right) \leftarrow \left(\begin{array}{c} p_0 \\ \hline \pi_1 \\ \hline p_2 \end{array}\right)$$

endwhile

Figure 1: LINPACK and LAPACK unblocked algorithms for the LU factorization, $\mathrm{LU}_{\mathrm{UNB}}^{\mathrm{LIN}}$ and $\mathrm{LU}_{\mathrm{UNB}}^{\mathrm{LAP}}$ respectively.

Algorithm: $[A,\ p] := [\{L\backslash U\},\ p] = \mathrm{LU}_{\mathrm{BLK}}(A)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$ and $p \rightarrow \left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right)$

where A_{TL} is 0×0 and p_T has 0 elements

while $n(A_{TL}) < n(A)$ **do**

Determine block size b

Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right) \text{ and } \left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right) \rightarrow \left(\begin{array}{c} p_0 \\ \hline p_1 \\ \hline p_2 \end{array}\right)$$

where A_{11} is $b \times b$ and p_1 has b elements

LINPACK variant:

$$\left[\left(\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right),\ p_1\right] := \left[\left(\begin{array}{c} \{L\backslash U\}_{11} \\ \hline L_{21} \end{array}\right),\ p_1\right] = \mathrm{LU}_{\mathrm{UNB}}^{\mathrm{LAP}}\left(\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right)$$

$\left(\begin{array}{c} A_{12} \\ \hline A_{22} \end{array}\right) := P(p_1)\left(\begin{array}{c} A_{12} \\ \hline A_{22} \end{array}\right)$

$A_{12} := U_{12} = L_{11}^{-1}A_{12}$

$A_{22} := A_{22} - L_{21}U_{12}$

LAPACK variant:

$$\left[\left(\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right),\ p_1\right] := \left[\left(\begin{array}{c} \{L\backslash U\}_{11} \\ \hline L_{21} \end{array}\right),\ p_1\right] = \mathrm{LU}_{\mathrm{UNB}}^{\mathrm{LAP}}\left(\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right)$$

$\left(\begin{array}{c|c} A_{10} & A_{12} \\ \hline A_{20} & A_{22} \end{array}\right) := P(p_1)\left(\begin{array}{c|c} A_{10} & A_{12} \\ \hline A_{20} & A_{22} \end{array}\right)$

$A_{12} := U_{12} = L_{11}^{-1}A_{12}$

$A_{22} := A_{22} - L_{21}U_{12}$

Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right) \text{ and } \left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right) \leftarrow \left(\begin{array}{c} p_0 \\ \hline p_1 \\ \hline p_2 \end{array}\right)$$

endwhile

Figure 2: LINPACK and LAPACK blocked algorithms for the LU factorization built upon an LAPACK unblocked factorization, $\mathrm{LU}_{\mathrm{BLK}}^{\mathrm{LIN}}$ and $\mathrm{LU}_{\mathrm{BLK}}^{\mathrm{LAP}}$ respectively.

However, in order to do so, the LAPACK algorithm requires the row permutations to be applied to both L and A [16]. Both of these unblocked algorithms are given in Figure 1. There, the notation

$$\left[\left(\frac{\alpha_{11}}{a_{21}}\right)\pi_1\right] := \textsc{Pivot}\left(\frac{\alpha_{11}}{a_{21}}\right)$$

refers to a function which swaps α_{11} and the element of largest magnitude in the input vector, and returns the index of that element in π_1; $P(\pi_1)$ denotes the corresponding permutation matrix.

Blocked variants of these algorithms cast the bulk of their computation in terms of matrix-matrix multiplication and inherently attain high performance on modern architectures (see, e.g., [24]). They are presented in Figure 2. In both algorithms, $P(p_1)$ refers to a permutation matrix obtained during the LU factorization with partial pivoting of the current column block.

3 Updating an LU factorization

In this section we discuss how to compute the LU factorization of a matrix A of the form

$$A = \left(\begin{array}{c|c} B & C \\ \hline D & E \end{array}\right) \tag{2}$$

in such a way that the LU factorization with partial pivoting of B can be reused if D, C, and E change. We consider A in (2) to be of dimension $n \times n$, with square B and E of orders n_B and n_E, respectively. For reference, factoring the matrix in (2) using the standard LU factorization with partial pivoting costs $\frac{2}{3}n^3$ floating-point arithmetic operations (FLOPs). Note that in this expression (and future computational cost estimates) we neglect terms of lower-order complexity, including the cost of pivoting the rows.

3.1 Basic procedure

We propose employing the following procedure, consisting of 5 steps, which computes an *LU factorization with incremental pivoting* of the matrix in (2):

Step 1: Factor B. Compute the LU factorization with partial pivoting

$$[B, p] := [\{L\backslash U\}, p] = \mathrm{LU}^{\mathrm{LAP}}_{\mathrm{BLK}}(B).$$

This step is skipped if B was already factored. If the factors are to be used for future updates to C, D, and E, then a copy of U is needed since it is overwritten during subsequent steps.

Step 2: Update C consistent with the factorization of B (by means of the forward substitution):

$$C := \mathrm{FS}^{\mathrm{LAP}}(B, p, C) = L^{-1}P(p)C.$$

Step 3: Factor $\left(\frac{U}{D}\right)$. Compute the LU factorization with partial pivoting

$$\left[\left(\frac{U}{D}\right), \bar{L}, r\right] := \left[\left(\frac{\{\bar{L}\backslash\bar{U}\}}{\tilde{L}}\right), r\right] = \mathrm{LU}^{\mathrm{LAP}}_{\mathrm{BLK}}\left(\frac{U}{D}\right).$$

Here $\bar{U}$ overwrites the upper triangular part of B (where U was stored before this operation). The lower triangular matrix $\bar{L}$ that results needs to be stored separately, since both L, computed in Step 1 and used at Step 2, and $\bar{L}$ are needed during the forward substitution stage when solving a linear system.

Step 4: Update $\left(\frac{C}{E}\right)$ consistent with the factorization of $\left(\frac{U}{D}\right)$:

$$\left(\frac{C}{E}\right) := \mathrm{FS}^{\mathrm{LAP}}\left(\left(\frac{\bar{L}}{D}\right), r, \left(\frac{C}{E}\right)\right).$$

Operation	Approximate cost (in flops)		
	Basic procedure	Structure-Aware LAPACK procedure	Structure-Aware LINPACK procedure
1: Factor B	$\frac{2}{3}n_B^3$	$\frac{2}{3}n_B^3$	$\frac{2}{3}n_B^3$
2: Update C	$n_B^2 n_E$	$n_B^2 n_E$	$n_B^2 n_E$
3: Factor $\begin{pmatrix} U \\ D \end{pmatrix}$	$n_B^2 n_E$ $+\frac{2}{3}n_B^3$	$n_B^2 n_E$ $+\frac{1}{2}bn_B^2$	$n_B^2 n_E$ $+\frac{1}{2}bn_B^2$
4: Update $\begin{pmatrix} C \\ E \end{pmatrix}$	$2n_B n_E^2$ $+n_B^2 n_E$	$2n_B n_E^2$ $+n_B^2 n_E$	$2n_B n_E^2$ $+bn_B n_E$
5: Factor E	$\frac{2}{3}n_E^3$	$\frac{2}{3}n_E^3$	$\frac{2}{3}n_E^3$
Total	$\frac{2}{3}n^3$ $+\frac{2}{3}n_B^3 + n_B^2 n_E$	$\frac{2}{3}n^3$ $+n_B^2\left(\frac{1}{2}b + n_E\right)$	$\frac{2}{3}n^3$ $+bn_B\left(\frac{n_B}{2} + n_E\right)$

Figure 3: Computational cost (in FLOPs) of the different approaches to compute the LU factorization of the matrix in (2). The highlighted costs are those incurred in excess of the cost of a standard LU factorization.

Step 5: Factor E. Finally, compute the LU factorization with partial pivoting

$$[E, s] := \left[\{\tilde{L}\backslash\tilde{U}\}, s\right] = \mathrm{LU}^{\mathrm{LAP}}_{\mathrm{BLK}}(E).$$

Overall, the five steps of the procedure apply Gauss transforms and permutations to reduce A to an upper triangular matrix as follows:

$$\left(\begin{array}{c|c} I & 0 \\ \hline 0 & \check{L}^{-1}P(s) \end{array}\right)\left(\begin{array}{c|c} \bar{L} & 0 \\ \hline \check{L} & I \end{array}\right)^{-1} P(r) \underbrace{\left(\begin{array}{c|c} L^{-1}P(p) & 0 \\ \hline 0 & I \end{array}\right)\left(\begin{array}{c|c} B & C \\ \hline D & E \end{array}\right)}_{\text{Steps 1 and 2}} =$$

$$\left(\begin{array}{c|c} I & 0 \\ \hline 0 & \check{L}^{-1}P(s) \end{array}\right) \underbrace{\left(\begin{array}{c|c} \bar{L} & 0 \\ \hline \check{L} & I \end{array}\right)^{-1} P(r)\left(\begin{array}{c|c} U & \hat{C} \\ \hline D & E \end{array}\right)}_{\text{Steps 3 and 4}} =$$

$$\underbrace{\left(\begin{array}{c|c} I & 0 \\ \hline 0 & \check{L}^{-1}P(s) \end{array}\right)\left(\begin{array}{c|c} \bar{U} & \check{C} \\ \hline 0 & \check{E} \end{array}\right)}_{\text{Step 5}} = \left(\begin{array}{c|c} \bar{U} & \bar{C} \\ \hline 0 & \tilde{U} \end{array}\right),$$

where $\{L\backslash U\}$, $\left\{\left(\begin{array}{c|c} \bar{L} & 0 \\ \hline \check{L} & I \end{array}\right) \backslash \begin{pmatrix} \bar{U} \\ 0 \end{pmatrix}\right\}$, and $\{\tilde{L}\backslash\tilde{U}\}$ are the triangular factors computed, respectively, in the LU factorizations in Steps 1, 3, and 5; and p, r, and s are the corresponding permutation vectors.

3.2 Analysis of the basic procedure

For now, the factorization in Step 3 does not take advantage of any zeroes below the diagonal of U: After matrix B is factored and C is updated, the matrix $\left(\begin{array}{c|c} U & C \\ \hline D & E \end{array}\right)$ is factored as if it were a matrix without special structure. Its computational cost is stated in the column labeled "Basic procedure" in Table 3. In this table, we assume that $b \ll n_E, n_B$ and report only those costs that equal at least $O(bn_E n_B)$, $O(bn_E^2)$, or $O(bn_B^2)$. *If n_E is small* (that is, $n_B \approx n$), the procedure clearly does not benefit from the existence of an already factored B. Also, the procedure requires additional storage for the $n_B \times n_B$ lower triangular matrix $\bar{L}$ computed in Step 3.

We describe next how to reduce both the computational and storage requirements by exploiting the upper triangular structure of U during Steps 3 and 4.

Algorithm: $\left[\left(\frac{U}{D}\right), \bar{L}, r\right] := \mathrm{LU}_{\mathrm{BLK}}^{\mathrm{SA-LIN}}\left(\frac{U}{D}\right)$

Partition $U \rightarrow \left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right)$, $D \rightarrow \left(\begin{array}{c|c} D_L & D_R \end{array}\right)$, $\bar{L} \rightarrow \left(\frac{L_T}{L_B}\right)$, $r \rightarrow \left(\frac{r_T}{r_B}\right)$

where U_{TL} is 0×0, D_L has 0 columns, $\bar{L}_T$ has 0 rows, and r_T has 0 elements

while $n(U_{TL}) < n(U)$ **do**

Determine block size b

Repartition

$$\left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} U_{00} & U_{01} & U_{02} \\ \hline 0 & U_{11} & U_{12} \\ \hline 0 & 0 & U_{22} \end{array}\right), \left(\begin{array}{c|c} D_L & D_R \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} D_0 & D_1 & D_2 \end{array}\right),$$

$$\left(\frac{\bar{L}_T}{\bar{L}_B}\right) \rightarrow \left(\frac{\bar{L}_0}{\bar{L}_1}{\bar{L}_2}\right), \left(\frac{r_T}{r_B}\right) \rightarrow \left(\frac{r_0}{r_1}{r_2}\right)$$

where U_{11} is $b \times b$, D_1 has b columns, $\bar{L}_1$ has b rows, and r_1 has b elements

---

$\left[\left(\frac{\{\bar{L}_1 \backslash U_{11}\}}{D_1}\right), r_1\right] := \mathrm{LU}_{\mathrm{UNB}}^{\mathrm{LAP}}\left(\frac{U_{11}}{D_1}\right)$

$\left(\frac{U_{12}}{D_2}\right) := P(r_1)\left(\frac{U_{12}}{D_2}\right)$

$U_{12} := \bar{L}_1^{-1} U_{12}$

$D_2 := D_2 - D_1 U_{12}$

Continue with

$$\left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} U_{00} & U_{01} & U_{02} \\ \hline 0 & U_{11} & U_{12} \\ \hline 0 & 0 & U_{22} \end{array}\right), \left(\begin{array}{c|c} D_L & D_R \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} D_0 & D_1 & D_2 \end{array}\right),$$

$$\left(\frac{\bar{L}_T}{\bar{L}_B}\right) \leftarrow \left(\frac{\bar{L}_0}{\bar{L}_1}{\bar{L}_2}\right), \left(\frac{r_T}{r_B}\right) \leftarrow \left(\frac{r_0}{r_1}{r_2}\right)$$

endwhile

Figure 4: SA-LINPACK blocked algorithm for the LU factorization of $\left(U^T,\ D^T\right)^T$ built upon an LAPACK blocked factorization.

3.3 Exploiting the structure in Step 3

A blocked algorithm that exploits the upper triangular structure of U during the factorization of $\left(\frac{U}{D}\right)$ is given in Figure 4 and illustrated in Figure 5. We name this algorithm $\mathrm{LU}_{\mathrm{BLK}}^{\mathrm{SA-LIN}}$ to reflect that it computes a "Structure-Aware" (SA) LU factorization. At each iteration of the algorithm, the panel of b columns consisting of $\left(\frac{U_{11}}{D_1}\right)$ is factored using the unblocked LAPACK algorithm $\mathrm{LU}_{\mathrm{UNB}}^{\mathrm{LAP}}$. (In our implementation this algorithm is modified to take advantage of the zeroes below the diagonal of U_{11}.) As part of the factorization, U_{11} is overwritten by $\{\bar{L}_1 \backslash \bar{U}_{11}\}$. However, in order to preserve the strictly lower triangular part of U_{11} (which contains the part of the matrix L that was computed in Step 1), we employ the $b \times b$ submatrix $\bar{L}_1$ of the $n_B \times b$ array $\bar{L}$ (see Figure 5). As in the blocked LINPACK algorithm in Figure 2, the LAPACK and LINPACK styles of pivoting are combined: the current panel of columns are pivoted using the LAPACK approach but the permutations from this factorization are only applied to $\left(\frac{U_{12}}{D_2}\right)$.

Figure 3 gives the cost of this approach in Step 3 of the column labeled "SA LINPACK procedure". The cost difference comes from the updates of U_{12} in Figure 4 which, provided $b \ll n_B$, is insignificant compared to $\frac{2}{3}n^3$.

A blocked SA LAPACK algorithm for Step 3 only differs from that in Figure 4 in that, at a certain iteration, after the LU factorization of the current panel is computed, these permutations must also be applied to $\left(\frac{U_{10}}{D_0}\right)$. As indicated in Step 3 of the column labeled "SA LAPACK procedure", this does not

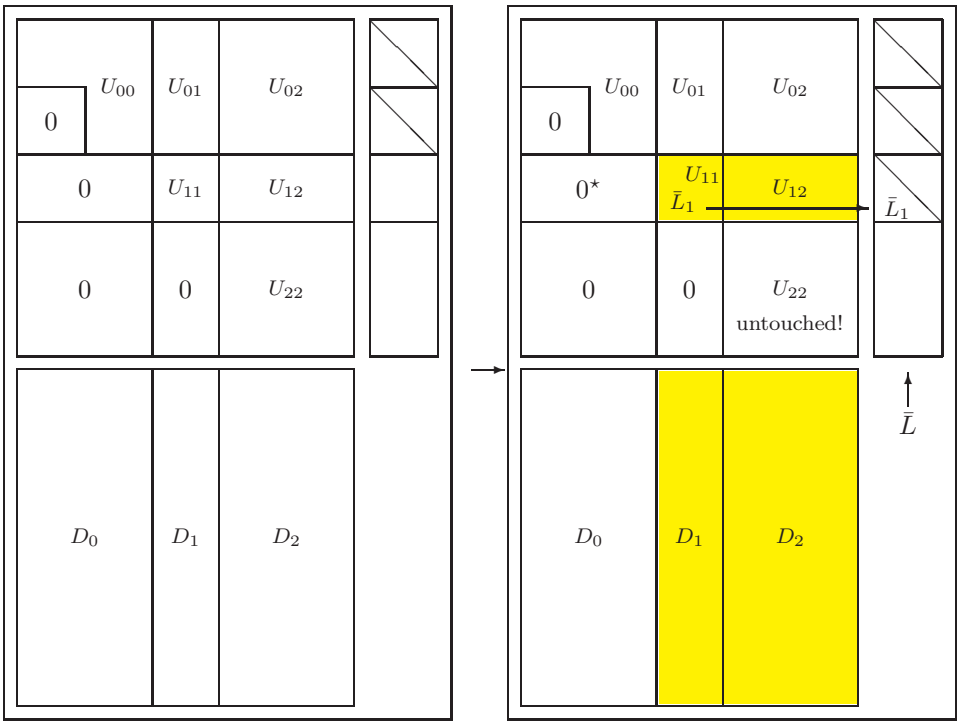

Figure 5: Illustration of an iteration of the SA LINPACK blocked algorithm used in Step 3 and how it preserves most of the zeroes in U. The zeroes below the diagonal are preserved, except within the $b \times b$ diagonal blocks, where pivoting will fill below the diagonal. The shaded areas are the ones updated as part of the current iteration. The fact that U_{22} is not updated demonstrates how computation can be reduced. If the SA LAPACK blocked algorithm were used, then nonzeroes would appear during this iteration in the block marked as $0^\star$, due to pivoting; as a result, upon completion, zeros would be lost in the full strictly lower triangular part of U.

incur a (significative) computational overhead for *this step*. However, it does require an $n_B \times n_B$ array for storing $\bar{L}$ (see Figure 5) and, as we will see next, makes Step 4 more expensive.

3.4 Revisiting the update in Step 4

The same optimizations made in Step 3 must now be carried over to the update of $\begin{pmatrix} C \\ E \end{pmatrix}$. The algorithm for this is given in Figure 6. Since computation corresponding to zeroes is avoided, the update may be performed with $2n_B n_E^2 + bn_B n_E$ FLOPs, as indicated in Step 4 of the column labeled as "SA LINPACK procedure" of Figure 3.

The blocked SA LAPACK algorithm in Step 3 destroys the structure of the lower triangular matrix, which cannot be recovered during the forward substitution stage in Step 4, and therefore incurs a significant additional computational cost, as reported in the column labeled as "SA LAPACK procedure" in Figure 3.

3.5 Key contribution

The cost of the approaches analyzed in Figure 3 is illustrated in Figure 7. Each curve reports, as a function of n_E, the ratio between the cost of that procedure and the cost of the LU factorization with partial pivoting for a matrix with $n_B = 1000$ with $b = 32$. The analysis shows that the overhead of the SA LINPACK procedure is consistently low. On the other hand, as $n_E/n \to 1$ the cost of the basic procedure, which is initially twice as expensive as that of the LU factorization with partial pivoting, decreases. The SA LAPACK procedure

Algorithm: $\left(\frac{C}{E}\right) := \mathrm{FS}^{\mathrm{SA-LIN}}_{\mathrm{BLK}}\left(\left(\frac{L}{D}\right), r, \left(\frac{C}{E}\right)\right)$

Partition $\bar{L} \rightarrow \left(\frac{L_T}{L_B}\right)$, $D \rightarrow \left(\begin{array}{c|c} D_L & D_R \end{array}\right)$, $r \rightarrow \left(\frac{r_T}{r_B}\right)$, $C \rightarrow \left(\frac{C_T}{C_B}\right)$,
where $\bar{L}_T$ and C_T have 0 rows, D_L has 0 columns, and r_T has 0 elements

while $n(D_L) < n(D)$ **do**
Determine block size b
Repartition

$$\left(\frac{\bar{L}_T}{L_B}\right) \rightarrow \left(\frac{\bar{L}_0}{\frac{L_1}{L_2}}\right), \left(\begin{array}{c|c} D_L & D_R \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} D_0 & D_1 & D_2 \end{array}\right),$$

$$\left(\frac{r_T}{r_B}\right) \rightarrow \left(\frac{r_0}{\frac{r_1}{r_2}}\right), \left(\frac{C_T}{C_B}\right) \rightarrow \left(\frac{C_0}{\frac{C_1}{C_2}}\right),$$

where $\bar{L}_1$ and C_1 have b rows, D_1 has b columns, and r_1 has b elements

---

$\left(\frac{C_1}{E}\right) := P(r_1)\left(\frac{C_1}{E}\right)$
$C_1 := \bar{L}_1^{-1} C_1$
$E := E - D_1 C_1$

Continue with

$$\left(\frac{\bar{L}_T}{L_B}\right) \leftarrow \left(\frac{\bar{L}_0}{\frac{L_1}{L_2}}\right), \left(\begin{array}{c|c} D_L & D_R \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} D_0 & D_1 & D_2 \end{array}\right),$$

$$\left(\frac{r_T}{r_B}\right) \leftarrow \left(\frac{r_0}{\frac{r_1}{r_2}}\right), \left(\frac{C_T}{C_B}\right) \leftarrow \left(\frac{C_0}{\frac{C_1}{C_2}}\right),$$

endwhile

Figure 6: SA-LINPACK blocked algorithm for the update of $\left(C^T,\ E^T\right)^T$ consistent with the SA-LINPACK blocked LU factorization of $\left(U^T,\ D^T\right)^T$.

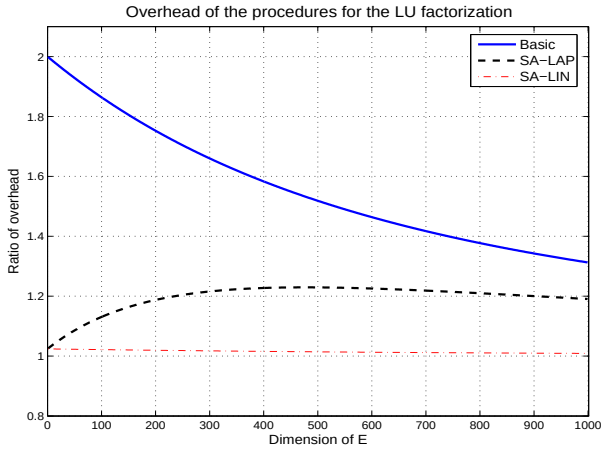

Figure 7: Overhead cost of the different approaches to compute the LU factorization in (2) with respect to the cost of the LU factorization with partial pivoting.

incurs only a negligible overhead as $n_E \rightarrow 0$ (that is, when the dimension of the update is very small).

The key insight of the proposed approach is the recognition that combining LINPACK and LAPACK styles of pivoting allows one to use a blocked algorithm while avoiding filling most of the zeroes in the lower

Step	Algorithm	Cost
	for $k = 0 : N-1$	
T-1	$[A_{kk}, p_{kk}] := \mathrm{LU}^{\mathrm{LAP}}_{\mathrm{BLK}}(A_{kk})$	$\frac{2}{3}t^3$ flops
	for $j = k+1 : N-1$	
T-2	$A_{kj} := \mathrm{FS}^{\mathrm{LAP}}(A_{kk}, p_{kk}, A_{kj})$	t^3 flops
	endfor	
	for $i = k+1 : N-1$	
T-3	$\left[\left(\frac{A_{kk}}{A_{ik}}\right), L_{ik}, p_{ik}\right] := \mathrm{LU}^{\mathrm{SA-LIN}}_{\mathrm{BLK}}\left(\frac{A_{kk}}{A_{ik}}\right)$	t^3 flops
	for $j = k+1 : N-1$	
T-4	$\left[\left(\frac{A_{kj}}{A_{ij}}\right)\right] := \mathrm{FS}^{\mathrm{SA-LIN}}_{\mathrm{BLK}}\left(\left(\frac{L_{ik}}{A_{ik}}\right), p_{ik}, \left(\frac{A_{kj}}{A_{ij}}\right)\right)$	$2t^3$ flops
	endfor	
	endfor	
	endfor	

Figure 8: Algorithm-by-blocks for the LU factorization with incremental pivoting. (Only leading term of cost of each operation is listed.)

triangular part of U. This, in turn, makes the extra cost of Step 4 acceptable. In other words, for the SA LINPACK procedure, the benefit of the higher performance of the blocked algorithm comes at the expense of a lower-order amount of extra computation.

The extra memory required by the SA LINPACK procedure consists of an $n_B \times n_B$ upper triangular matrix and an $n_B \times b$ array.

4 An Algorithm-By-Blocks

In this section it is shown how the insights from the previous section can be used to implement an algorithm-by-blocks for the LU factorization with incremental pivoting. Throughout this section we will consider a matrix A of dimension $n \times n$.

Assume for simplicity that $n = Nt$, where N is an integer, and consider the partitioning by *tiles*

$$A \rightarrow \left(\begin{array}{c|c|c} A_{00} & \cdots & A_{0,N-1} \\ \hline \vdots & \ddots & \vdots \\ \hline A_{N-1,0} & \cdots & A_{N-1,N-1} \end{array}\right),$$

with all A_{ij} of size $t \times t$. Then the algorithm in Figure 8 is a generalization of the algorithm described in Section 3 that computes the LU factorization of A with incremental pivoting. The algorithm is annotated with the cost of each operation in terms of FLOPs.

The total number of FLOPs performed by the algorithm-by-blocks is approximately given by

$$\sum_{k=0}^{N-1}\left(\frac{2}{3}t^2 + \sum_{j=k+1}^{N-1} t^3 + \sum_{i=k+1}^{N-1}\left(t^3 + \sum_{j=k+1}^{N-1} 2t^3\right)\right) \approx \frac{2}{3}\left(\frac{n}{t}\right)^3 t^3 = \frac{2}{3}n^3.$$

Notice that that there is some flexibility in the order in which the loops are arranged. Indeed, the SuperMatrix runtime system, described in Section 6.3, rearranges the operations and therefore the exact order of the loops is not important.

In [6] it is claimed that a similar algorithm-by-blocks requires 50% additional FLOPs. Our analysis shows this overhead can be avoided.

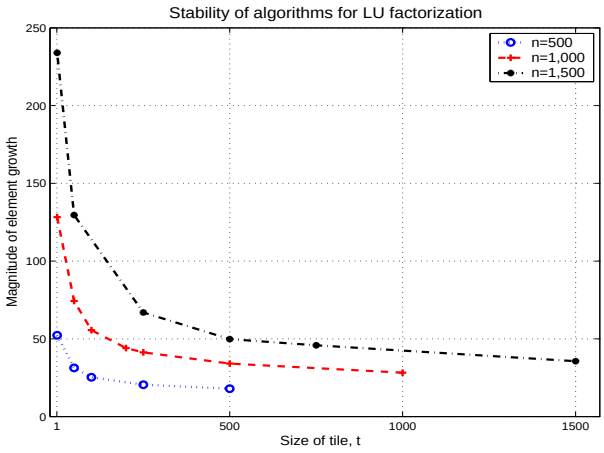

Figure 9: Element growth in the LU factorization using different pivoting techniques.

5 Remarks on the Numerical Stability

The algorithm-by-blocks for the LU factorization with incremental pivoting carries out a sequence of row permutations (corresponding to the application of pivots) which are different from those that would be performed in an LU factorization with partial pivoting. Therefore, the numerical stability of this algorithm is also different. In this section we offer some remarks on the stability of the algorithm-by-blocks.

The numerical (backward) stability of an algorithm that computes the LU factorization of a matrix A depends on the growth factor [34]

$$\rho = \frac{\|L\|\|U\|}{\|A\|}, \tag{3}$$

which is basically determined by the problem size and the pivoting strategy. For example, the growth factors of complete, partial, and *pairwise* ([39, p. 236]) pivoting have been demonstrated to be bounded as $\rho_c \leq n^{1/2}(2 \cdot 3^{1/2} \cdots n^{1/n-1})$, $\rho_p \leq 2^{n-1}$, and $\rho_w \leq 4^{n-1}$, respectively [32, 34]. Statistical models and extensive experimentation in [36] have shown that, on average, $\rho_c \approx n^{1/2}$, $\rho_p \approx n^{2/3}$, and $\rho_w \approx n$. Thus, in practice partial/pairwise pivoting are both numerically stable and pairwise pivoting can be expected to numerically behave only slightly worse than partial pivoting.

The algorithm-by-blocks applies partial pivoting during the factorizations in steps T-1 and T-3. Furthermore, tiles are annihilated pairwise in what can be considered a blocked (or tiled) version of pairwise pivoting. Thus, we can expect an element growth for the algorithm-by-blocks that is between those of partial and pairwise pivoting. In particular, if the tile size equals the problem size ($t = n$) our algorithm strictly employs partial pivoting, while if $t = 1$ the algorithm employs pairwise pivoting. Next we discuss the results of an experiment that provides evidence in support of this observation.

In Figure 9 we report the element growths observed during the computation of the LU factorization of matrices of dimensions $n = 500$, 1000, and 1500, using partial ($t = n$), incremental ($1 < t < n$), and pairwise pivoting ($t = 1$). The entries of the matrices are generated randomly, chosen from a uniform distribution in the interval $(0.0, 1.0)$. The experiment was carried out on an Intel Xeon processor using MATLAB® 7.0.0 (IEEE double-precision arithmetic). The results report the average element growth for 20 different matrices for each matrix dimension. The figure shows that the growth factor of incremental pivoting is smaller than that of pairwise pivoting and approximates that of partial pivoting as the tile size increases.

For those who are not sufficiently satisfied with the element growth of incremental pivoting, we suggest performing a few refinement iterations of the solution to $Ax = b$ as this guarantees stability at a low computational cost [22]. We can combine this strategy with an estimation of the backward error $\|PA - LU\|_1$ *a posteriori*, at a cost of $O(n^2)$ FLOPs, to determine whether iterative refinement is actually needed.

6 Tools

In this section we briefly review some of the tools that the FLAME project puts at our disposal.

6.1 The FLAME/C API

The algorithms presented in this paper require intricate modifications to the standard implementations. The FLAME/C API allows the algorithms that are given in Figures 4 and 6 to be coded in the C programming language such that the code closely reflects these algorithm [4], thereby greatly reducing the time required for development of library routines.

6.2 FLASH

While many have studied the benefits of storing matrices by blocks [10, 15, 21, 28], the conventional approach inherently engenders complex and unwieldy code implementations. Some researchers have tried to solve this via programming language solutions [37, 40] while others view matrices as higher dimensional arrays to capture the levels of blocking [1, 19].

We have observed that, conceptually, one naturally thinks of a matrix stored by blocks as a matrix of submatrices. As a result, if the API encapsulates information that describes a matrix in an object, as FLAME/C does, and allows an element in a matrix to itself be a matrix object, then algorithms over matrices stored by blocks can be represented in code at the same high level of abstraction. This layering may be instantiated recursively if multiple levels of hierarchy in the matrix are to be exposed. We call this extension to FLAME/C the *FLASH* API [27]. Examples of how simpler operations can be transformed from FLAME to FLASH implementations may be found in [7, 9].

6.3 SuperMatrix

Finally, we observed that, given an API that views matrices as composed of unit blocks and an algorithm implemented using this API, the inner workings of the library can be changed so that instead of executing operations over blocks, sub-operations can be enqueued as tasks and subsequently assembled into a directed acyclic graph (DAG) that represents dependencies between sub-operations. The DAG can then be exploited by a runtime system that dynamically schedules tasks for execution as dependencies are fulfilled. These two phases—constructing the DAG (*analyzer*) and scheduling the tasks (*scheduler/dispatcher*)—can take place transparently regardless of the algorithm used in the library routine.

To accomplish this, the subproblems within the sequential algorithm are replaced (via C preprocessor macros) with function invocations that enqueue all pertinent information about the sub-operation on a global task queue. Once all tasks are enqueued, the DAG is complete, and a separate function call initiates parallel execution. When a task completes execution, all dependent tasks that use blocks updated by the recently completed task are "notified". Once a notified task has all of its dependencies fulfilled, it is marked as ready and available and then enqueued at the tail of the *waiting queue*. Idle threads dequeue tasks from the head of this second queue until all tasks have been executed. We call this extension to FLASH the *SuperMatrix* runtime system since it allows out-of-order computation similar to machine instructions within superscalar architectures [20]. For further details on SuperMatrix, see [7, 9, 30, 8, 31].

7 Experiments

In this section, we examine the three approaches for the LU factorization in order to assess the potential performance benefits offered by the algorithm-by-blocks.

All experiments were performed on an SGI Altix 350 server using double-precision floating-point arithmetic. This ccNUMA architecture consists of eight nodes, each with two 1.5 GHz Intel Itanium2 processors, providing a total of 16 CPUs and a peak performance of 96 GFLOPs/sec. (96×10^9 floating-point operations per second). The nodes are connected via an SGI NUMAlink connection ring and collectively provide 32 GB

(32×2^{30} bytes) of general-purpose physical RAM. The OpenMP implementation provided by the Intel C Compiler served as the underlying threading mechanism used by SuperMatrix. Performance was measured by linking to the BLAS in Intel Math Kernel Library (MKL) version 8.1.

We report the performance of the following four parallel implementations of the LU factorization:

- **LAPACK `dgetrf` + multithreaded MKL**. LAPACK 3.0 routine `dgetrf` (LU factorization) linked to multithreaded BLAS in MKL 8.1.
- **Multithreaded MKL `dgetrf`**. Multithreaded implementation of routine `dgetrf` (LU factorization) in MKL 8.1.
- **AB + serial MKL**. Our implementation of the algorithm-by-blocks, with matrices stored in traditional column-major order so that blocks are not contiguous, tasks scheduled using the SuperMatrix run-time system, and linked to serial BLAS in MKL 8.1.
- **AB + serial MKL + contiguous blocks**. Our implementation of the algorithm-by-blocks, with matrices stored in contiguous blocks, tasks scheduled using the SuperMatrix run-time system, and linked to serial BLAS in MKL 8.1.

Remarkable performance results are reported in Figure 10. The matrix size ($m = n$) is reflected along the x-axis and the y-axis is scaled such that the top of the graph represents the theoretical peak performance of the system. When hand-tuning block sizes, an effort was made to determine the best values of inner blocking, b and outer blocking t for all combinations of parallel implementations and BLAS. The algorithmic block size used for LAPACK and parameter t for the indicated algorithm-by-blocks are reported in Figure 11. An inner block size of 16 was used for all problem sizes by both algorithm-by-blocks implementations. The block size used by MKL implementation of `dgetrf` is internally hidden in the library and unknown to us at the time of this writing.

8 Conclusions

We have shown that an algorithm-by-blocks first developed for out-of-core computation can be easily converted to a parallel algorithm that targets multithreaded architectures. By executing the algorithm with the SuperMatrix runtime system on matrices stored by blocks, remarkable performance was attained relative to LAPACK and MKL `dgetrf` implementations. Empirical evidence indicates that the element growth due to incremental pivoting is likely to be reasonable, resulting in an algorithm that is only mildly less stable than the traditional LU factorization with partial pivoting.

With this study, we have demonstrated the benefits of algorithms-by-blocks, coupled with the SuperMatrix runtime system, for all three major factorization operations: Cholesky factorization [7], QR factorization [30], and now LU factorization. In addition, the benefits of algorithms-by-blocks and SuperMatrix for the parallel implementation of the level-3 BLAS were discussed in [9], the inversion of a symmetric positive definite matrix in [8], and the factorization of band matrices in [31]. Altogether, these papers suggest the broad applicability of this approach toward the goal of retargeting libraries such as LAPACK and FLAME to multithreaded architectures.

Possibly the most important contribution of this and previous related work is a practical demonstration of the reduced programming burden required for implementing algorithms such as the one discussed in this paper. With the tools provided by FLAME/C, FLASH, and SuperMatrix, the time required to take an algorithm from whiteboard to high-performance parallel implementation may be measured in days rather than weeks or months.

Acknowledgements

This research was partially sponsored by NSF grants CCF–0540926 and CCF–0702714. We thank the other members of the FLAME team for their support.

Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation (NSF).

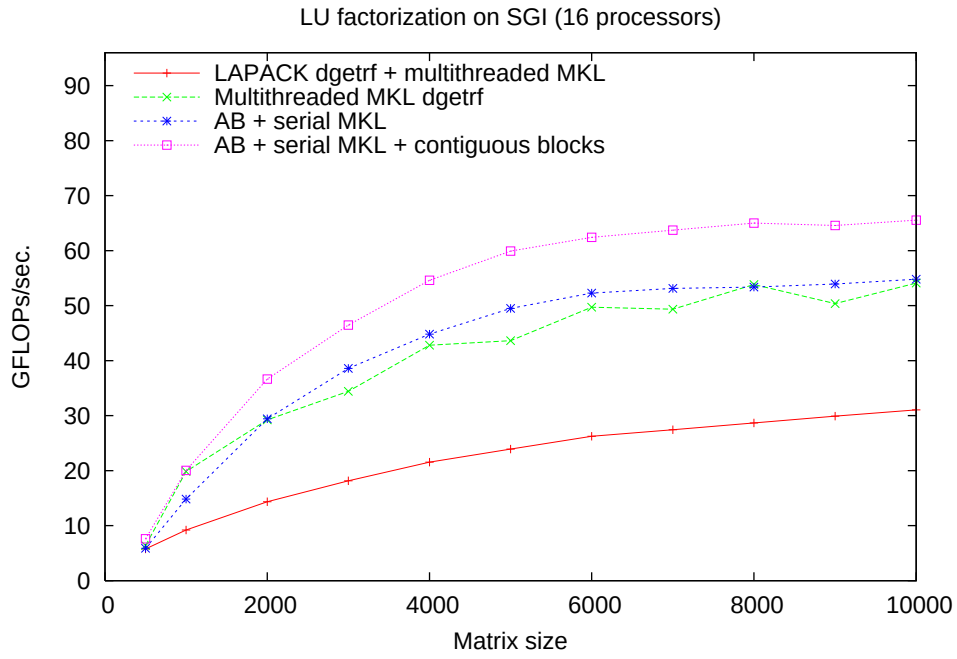

Figure 10: Performance of the LU factorization algorithms on 16 CPUs.

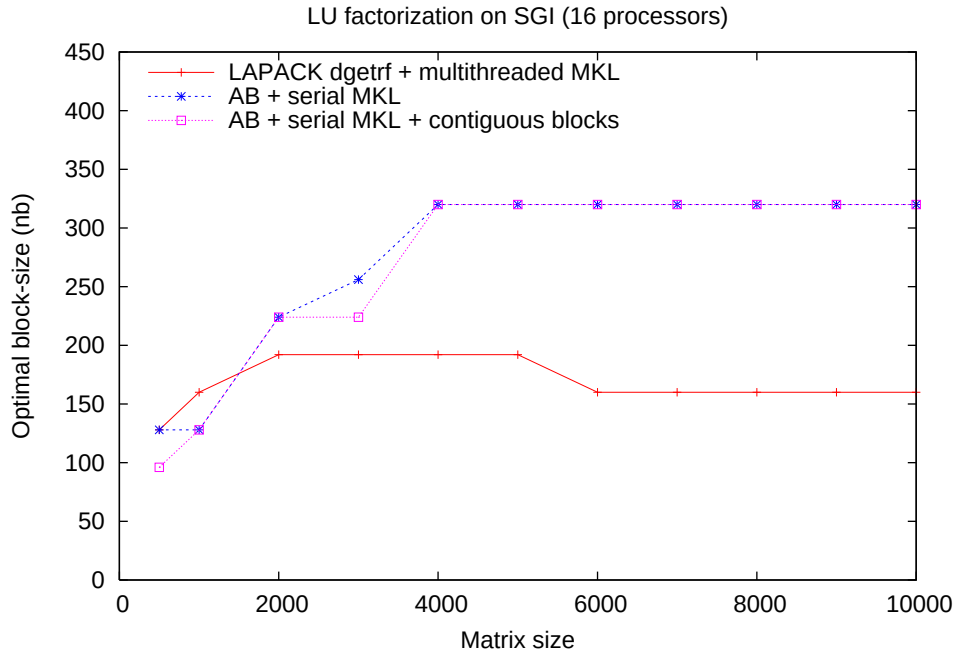

Figure 11: Block sizes used by the different implementations.

References

[1] R. C. Agarwal and F. G. Gustavson. Vector and parallel algorithms for Cholesky factorization on IBM 3090. In *Supercomputing '89: Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pages 225–233, New York, NY, USA, 1989. ACM Press.

[2] Pieter Bellens, Josep M. Pérez, Rosa M. Badía, and Jesús Labarta. CellSs: a programming model for the Cell BE architecture. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 86, New York, NY, USA, 2006. ACM Press.

[3] Paolo Bientinesi, John A. Gunnels, Margaret E. Myers, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. The science of deriving dense linear algebra algorithms. *ACM Transactions on Mathematical Software*, 31(1):1–26, March 2005.

[4] Paolo Bientinesi, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. Representing linear algebra algorithms in code: The FLAME application programming interfaces. *ACM Trans. Math. Soft.*, 31(1):27–59, March 2005.

[5] Afredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. LAPACK Working Note 191 UT-CS-07-600, University of Tennessee, September 2007.

[6] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. Parallel tiled QR factorization for multicore architectures. LAPACK Working Note 190 UT-CS-07-598, University of Tennessee, July 2007.

[7] Ernie Chan, Enrique Quintana-Ortí, Gregorio Quintana-Ortí, and Robert van de Geijn. SuperMatrix out-of-order scheduling of matrix operations for SMP and multi-core architectures. In *SPAA '07: Proceedings of the Nineteenth ACM Symposium on Parallelism in Algorithms and Architectures*, pages 116–126, 2007.

[8] Ernie Chan, Field G. Van Zee, Paolo Bientinesi, Enrique S. Quintana-Ortí, Gregorio Quintana-Ortí, and Robert van de Geijn. Supermatrix: A multithreaded runtime scheduling system for algorithms-by-blocks. FLAME Working Note #25 TR-07-41, The University of Texas at Austin, Department of Computer Sciences, August 2007.

[9] Ernie Chan, Field Van Zee, Robert van de Geijn, Enrique S. Quintana-Ortí, and Gregorio Quintana-Ortí. Satisfying your dependencies with SuperMatrix. In *IEEE Cluster 2007*, pages 92–99, 2007.

[10] S. Chatterjee, A.R. Lebeck, P.K. Patnala, and M. Thottethodi. Recursive array layouts and fast matrix multiplication. *IEEE Trans. on Parallel and Distributed Systems*, 13(11):1105–1123, 2002.

[11] 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 Comput. Soc. Press, 1992.

[12] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart. *LINPACK Users' Guide*. SIAM, Philadelphia, 1979.

[13] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 16(1):1–17, March 1990.

[14] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 14(1):1–17, March 1988.

[15] Erik Elmroth, Fred Gustavson, Isak Jonsson, and Bo Kagstrom. Recursive blocked algorithms and hybrid data structures for dense matrix library software. *SIAM Review*, 46(1):3–45, 2004.

[16] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, 3rd edition, 1996.

[17] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. FLAME: Formal Linear Algebra Methods Environment. *ACM Trans. Math. Soft.*, 27(4):422–455, December 2001.

[18] Brian C. Gunter and Robert A. van de Geijn. Parallel out-of-core computation and updating the QR factorization. *ACM Transactions on Mathematical Software*, 31(1):60–78, March 2005.

[19] F. G. Gustavson, L. Karlsson, and B. Kagstrom. Three algorithms on distributed memory using packed storage. *Computational Science – Para 2006*. Bo Kagstrom, E. Elmroth, eds., accepted for Lecture Notes in Computer Science. Springer-Verlag, 2007.

[20] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufman, 3rd edition, 2003.

[21] Greg Henry. BLAS based on block data structures. Theory Center Technical Report CTC92TR89, Cornell University, Feb. 1992.

[22] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 2002.

[23] Thierry Joffrain, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. Rapid development of high-performance out-of-core solvers. In *PARA'04*, volume 3732 of *Lecture Notes in Computer Science*, pages 413–422. Springer-Verlag, 2005.

[24] B. Kågström, P. Ling, and C. Van Loan. GEMM-based level 3 BLAS: High-performance model, implementations and performance evaluation benchmark. LAPACK Working Note #107 CS-95-315, Univ. of Tennessee, Nov. 1995.

[25] James Kahle, Michael Day, Peter Hofstee, Charles Johns, Theodore Maeurer, and David Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, 49(4/5):589–604, September 2005.

[26] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Soft.*, 5(3):308–323, Sept. 1979.

[27] Tze Meng Low and Robert van de Geijn. An API for manipulating matrices stored by blocks. Technical Report TR-2004-15, Department of Computer Sciences, The University of Texas at Austin, May 2004.

[28] N. Park, B. Hong, and V.K. Prasanna. Tiling, block data layout, and memory hierarchy performance. *IEEE Trans. on Parallel and Distributed Systems*, 14(7):640–654, 2003.

[29] Enrique S. Quintana-Ortí and Robert van de Geijn. Updating an LU factorization with pivoting. *ACM Trans. Math. Soft.*, 2007. To appear.

[30] Gregorio Quintana-Ortí, Enrique Quintana-Ortí, Ernie Chan, Field G. Van Zee, and Robert van de Geijn. Scheduling of QR factorization algorithms on SMP and multi-core architectures. FLAME Working Note #24 TR-07-37, The University of Texas at Austin, Department of Computer Sciences, July 2007.

[31] Gregorio Quintana-Ortí, Enrique S. Quintana-Ortí, Alfredo Remón, and Robert van de Geijn. SuperMatrix for the factorization of band matrices. FLAME Working Note #27 TR-07-51, The University of Texas at Austin, Department of Computer Sciences, September 2007.

[32] Danny C. Sorensen. Analysis of pairwise pivoting in Gaussian elimination. *IEEE Trans. on Computers*, c-34(3):274–278, 1985.

[33] G. W. Stewart. *Introduction to Matrix Computations*. Academic Press, Orlando, Florida, 1973.

[34] G. W. Stewart. *Matrix Algorithms. Volume I: Basic Decompositions*. SIAM, Philadelphia, 1998.

[35] Sivan Toledo. A survey of out-of-core algorithms in numerical linear algebra. In James Abello and Jeffrey Scott Vitter, editors, *External Memory Algorithms and Visualization*, pages 161–180. American Mathematical Society Press, Providence, RI, 1999.

[36] Lloyd N. Trefethen and Robert S. Schreiber. Average-case stability of Gaussian elimination. *SIAM J. Matrix Anal. Appl.*, 11(3):335–360, 1990.

[37] Vinod Valsalam and Anthony Skjellum. A framework for high-performance matrix multiplication based on hierarchical abstractions, algorithms and optimized low-level kernels. *Concurrency and Computation: Practice and Experience*, 14(10):805–840, 2002.

[38] Robert A. van de Geijn. *Using PLAPACK: Parallel Linear Algebra Package*. The MIT Press, 1997.

[39] J. H. Wilkinson. *The Algebraic Eigenvalue Problem*. Oxford University Press, London, 1965.

[40] David S. Wise, Jeremy D. Frens, Yuhong Gu, and Gregory A. Alexander. Language support for morton-order matrices. In *PPoPP '01: Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, pages 24–33, New York, NY, USA, 2001. ACM Press.