# Design of Scalable Dense Linear Algebra Libraries for Multithreaded Architectures: the LU Factorization

Gregorio Quintana-Ortí[1], Enrique S. Quintana-Ortí[1], Ernie Chan[2], Robert A. van de Geijn[2], and Field G. Van Zee[2]

[1] Departamento de Ingeniería y Ciencia de Computadores, Universidad Jaume I, 12.071–Castellón, Spain, {gquintan,quintana}@icc.uji.es
[2] Department of Computer Sciences, The University of Texas at Austin, Austin, Texas 78712, {echan,field,rvdg}@cs.utexas.edu

**Abstract.** The scalable parallel implementation, targeting SMP and/or multicore architectures, of dense linear algebra libraries is analyzed. Using the LU factorization as a case study, it is shown that an algorithm-by-blocks exposes a higher degree of parallelism than traditional implementations based on multithreaded BLAS. The implementation of this algorithm using the SuperMatrix runtime system is discussed and the scalability of the solution is demonstrated on two different platforms with 16 processors.

**Key words:** Dense linear algebra libraries, high-level APIs, run-time system, multithreaded architectures, LU factorization.

## 1 Introduction

With the emergence of parallel computing architectures with many processing elements (e.g., SMP systems with dozens of processors, multicore chips with many cores, and CPUs featuring hardware accelerators such as the Cell processor [17, 1]), it is now widely recognized that commonly used dense linear algebra 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 [6–8, 20, 22, 5, 4].

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:

– A demonstration that dense linear algebra operations 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) [18, 10, 9], to the pure algorithm-by-blocks first proposed in [15]. This algorithm is similar to the algorithm-by-blocks for the QR factorization proposed in [13], for which multithreaded parallel implementations are given in [20, 5].
– Further examples of how (a) the FLASH extension of FLAME/C supports storage by blocks for this type of algorithms and (b) the SuperMatrix runtime system supports, transparent to the programmer, out-of-order computation on blocks.
– An analysis which reveals that the extra work associated with the algorithm-by-blocks represents a lower order cost term, in contrast to a claim in [5], and performance that rivals with that of an algorithms-by-blocks for the QR factorization.

The remainder of the paper is structured as follows: Sections 2 and 3, respectively, review the LAPACK algorithm for the LU factorization with partial pivoting and introduce our algorithm-by-blocks for the LU factorization with incremental pivoting. Section 4 provides an overview of various tools and methods derived from the FLAME project which were used in the implementation. Performance results can be found in Section 5 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 [12, 2]. 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.

## 2 The LU Factorization with Partial Pivoting

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

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

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 [11, 23]. Here, a sequence of permutation matrices and Gauss elimination matrices 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.

Blocked variants of the LU factorization cast the bulk of their computation in terms of matrix-matrix multiplication and inherently attain high performance on modern architectures (see, e.g., [16]). LAPACK unblocked and blocked algorithms for the LU factorization with partial pivoting are given in Figure 1. Provided the (inner) block size $b \ll n$, a major part of the computation in the blocked algorithm is cast in terms of the matrix product that updates $A_{22}$.

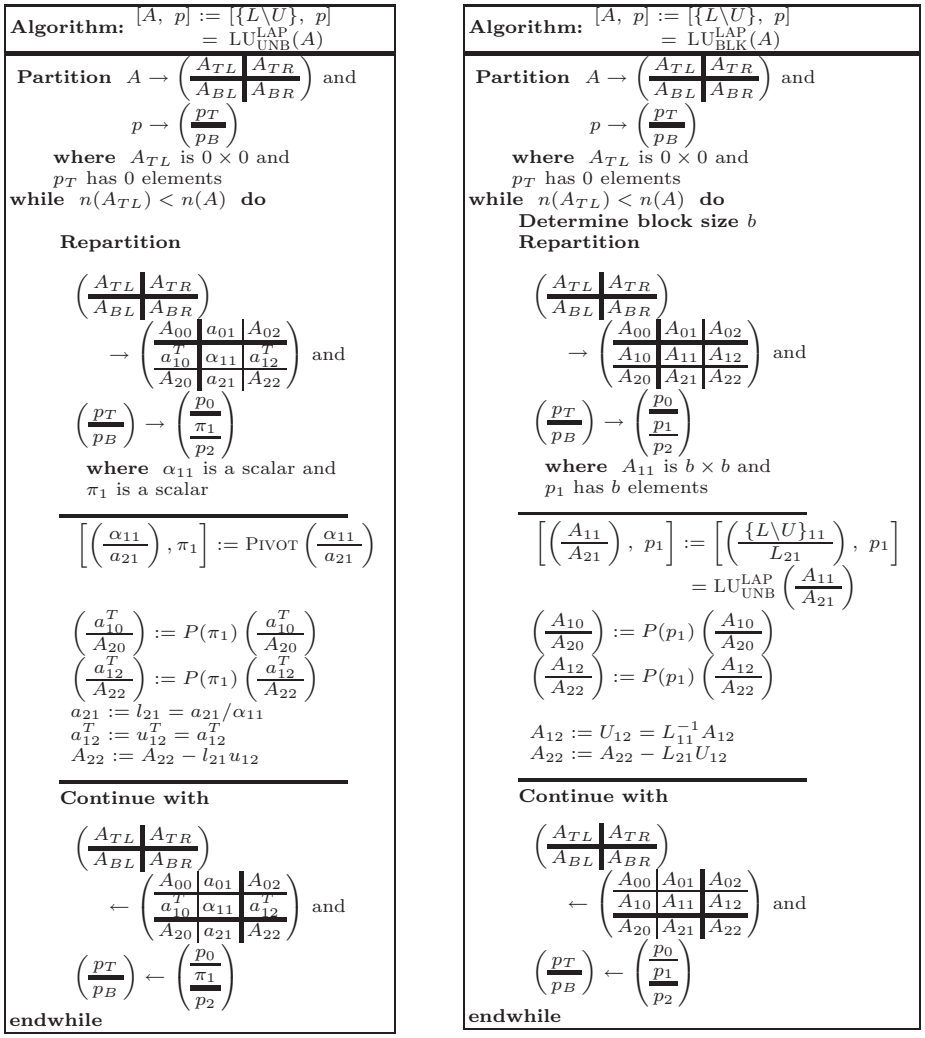


**Fig. 1.** LAPACK unblocked (left) and blocked (right) algorithms for the LU factorization, $\mathrm{LU}^{\mathrm{LAP}}_{\mathrm{UNB}}$ and $\mathrm{LU}^{\mathrm{LAP}}_{\mathrm{BLK}}$, respectively. $P(\pi_1)$ and $P(p_1)$ refer to permutation matrices. Function PIVOT swaps $\alpha_{11}$ and the element of largest magnitude in the input vector, and returns the index of that element in $\pi_1$.

Implementations of these algorithms, whether unblocked or blocked, are typically written to perform linear algebra operations, such as matrix-vector and matrix-matrix multiplication, via calls to the *Basic Linear Algebra Subprograms* (BLAS).

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

**Fig. 2.** Algorithm-by-blocks for the LU factorization with incremental pivoting and cost of the operations (only major terms of the cost are listed). $\mathrm{TRIU}(A_{kk})$ refers to the upper triangular part of the matrix.

Parallelism can be attained from these implementations within each invocation of a BLAS routine with the following benefits:

- The approach allows legacy libraries, such as LAPACK, to be used without modifying the library source code.
- Parallelism within sub-operations, e.g., the update of $A_{22}$, can be achieved through multithreaded implementations of the BLAS.

Disadvantages of this approach, on the other hand, include:

- The degree of parallelism achieved is potentially limited by the efficiency of the underlying multithreaded BLAS implementation.
- The end of each BLAS routine executed becomes an implicit synchronization point (or barrier) for the threads.
- For many operations the choice of algorithmic variant can significantly impact performance.

In the next section we propose an algorithm to overcome these difficulties.

## 3 An Algorithm-By-Blocks

In [21] it is shown how the insights gained from studying the problem of updating an existing LU factorization yields the algorithm-by-blocks for the LU factorization with incremental pivoting described next. (The algorithm was first introduced in [15], as an out-of-core algorithm-by-tiles.) 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

$$A \rightarrow \left( \begin{array}{c|c|c|c} A_{00} & A_{01} & \cdots & A_{0,N-1} \\ \hline A_{10} & A_{11} & \cdots & A_{1,N-1} \\ \hline \vdots & \vdots & \ddots & \vdots \\ \hline A_{N-1,0} & A_{N-1,1} & \cdots & A_{N-1,N-1} \end{array} \right),$$

with all $A_{ij}$ of size $t \times t$ ($t$ is the outer block size). Then, the algorithm in Figure 2, combined with the building blocks in Figures 1, 3 and 4, computes an *LU factorization of A with incremental pivoting*. The algorithm is annotated with the cost of each operation (building block) in terms of floating-point arithmetic operations (flops).

The key insight that allows the computational expense to be roughly the same as the standard LU factorization with pivoting lies with a careful orchestration of computation and pivoting so that the matrix on the diagonal, after being factored itself, does not incur fill-in as it is being used to zero elements in the blocks below it. For details, consult the algorithms in Figures 3 and 4, and [15, 21].

Provided $b \ll t$, $t \ll n$, and neglecting lower order terms, the total number of flops performed by the algorithm-by-blocks is approximately given by

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

In [5] it is claimed that a similar algorithm-by-blocks requires 50% additional flops. Our analysis shows this overhead can be avoided; see [21].

Notice that that there is some flexibility in the order in which the loops are arranged. Indeed, the SuperMatrix run-time system, described in the next section, rearranges the operations and therefore the exact order of the loops is not important.

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. An analysis of the stability of the algorithm-by-blocks is given in [21].

## 4 Tools

In this section we briefly review some of the tools that the FLAME project puts at our disposal: high level APIs to both code linear algebra algorithms (FLAME/C) and handle matrices stored by blocks (FLASH), and a run-time system to schedule tasks to execution as dependencies are fulfilled (SuperMatrix).

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

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

**where** $U_{TL}$ is $0 \times 0$, $D_L$ 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**

**Fig. 3.** SA-LINPACK blocked algorithm for the LU factorization of $\left(U^T,\ D^T\right)^T$ built upon an LAPACK blocked factorization.

The algorithm presented in this paper requires elaborated modifications to the standard implementations. The FLAME/C API allows the algorithms that are given in Figures 1, 3, and 4 to be coded in the C programming language such that the code closely reflects these algorithm [3], thereby greatly reducing the time required for development of library routines.

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

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

**Partition** $\bar{L} \rightarrow \left(\frac{\bar{L}_T}{\bar{L}_B}\right)$, $D \rightarrow \left(\begin{array}{c|c} D_L & D_R \end{array}\right)$, $r \rightarrow \left(\frac{r_T}{r_B}\right)$, $C \rightarrow \left(\frac{C_T}{C_B}\right)$,
**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**

$$\left(\frac{\bar{L}_T}{\bar{L}_B}\right) \rightarrow \left(\frac{\bar{L}_0}{\frac{\bar{L}_1}{\bar{L}_2}}\right), \left(\begin{array}{c|c} D_L & D_R \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} D_0 & D_1 & D_2 \end{array}\right),$$

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

**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**

$$\left(\frac{\bar{L}_T}{\bar{L}_B}\right) \leftarrow \left(\frac{\bar{L}_0}{\frac{\bar{L}_1}{\bar{L}_2}}\right), \left(\begin{array}{c|c} D_L & D_R \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} D_0 & D_1 & D_2 \end{array}\right),$$

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

**endwhile**

**Fig. 4.** SA-LINPACK blocked algorithm for the update of $\left(C^T,\ E^T\right)^T$ consistent with the SA-LINPACK blocked LU factorization of $\left(U^T,\ D^T\right)^T$.

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 [19]. Examples of how simpler operations can be transformed from FLAME to FLASH implementations can be found in [6, 8].

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 be-

tween sub-operations. The DAG can then be exploited by a run-time 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 calls to dense linear algebra kernels within the sequential algorithm are replaced 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 then enqueued at the tail of the *execution 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* run-time system since it allows out-of-order computation similar to machine instructions within superscalar architectures [14]. For further details on SuperMatrix, see [6–8, 20–22].

## 5 Experiments

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

All experiments were performed using double-precision floating-point arithmetic. Details on the platforms that were employed in the experimental evaluation are given in Table 1. Both architectures consist of a total of 16 CPUs: SET is a CC-NUMA platform with 16 processors while NEUMANN is an SMP of 8 processors with 2 cores each. The peak performance is 96 GFLOPS ($96 \times 10^9$ flops per second) for SET and 70.4 GFLOPS for NEUMANN.

| Platform | Architecture | Frequency (GHz) | L2 cache (KBytes) | L3 cache (MBytes) | Total RAM (GBytes) |
|---|---|---|---|---|---|
| SET | Intel Itanium2 | 1.5 | 256 | 4096 | 30 |
| NEUMANN | AMD Opteron | 2.2 | 1024 | – | 63 |

| Platform | Compiler | Optimization flags | BLAS | Operating System |
|---|---|---|---|---|
| SET | `icc` 9.0 | `-O3` | MKL 8.1 | Linux 2.6.5-7.244-sn2 |
| NEUMANN | `icc` 9.1 | `-O3` | MKL 9.1 | Linux 2.6.18-8.1.6.el5 |

**Table 1.** Architectures (top) and software (bottom) employed in the evaluation.

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.
- **Multithreaded MKL `dgetrf`**. Multithreaded implementation of routine `dgetrf` in MKL.
- **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.
- **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.

The GFLOPS rate attained by the different implementations are reported in Figure 5. A flop count of $2n^3$ flops was used for all algorithms. 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. An effort was made to determine the best values of the inner and outer block sizes, $b$ and $t$ respectively, for all combinations of parallel implementations and BLAS. An inner block size $b = 16$ was used for all problem dimensions in the algorithm-by-blocks implementation. The best outer block size $t$ was a function of the problem dimension, with values between 64 and 320 providing the best results. 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. The results show that the algorithm-by-blocks clearly outperforms the LAPACK implementation for all problem sizes on both platforms. Only on NEUMANN and starting from problem dimensions over 6,000, the multithreaded MKL implementation of `dgetrf` attains a higher performance than our algorithm-by-blocks.

## 6 Conclusions

With this study, we have demonstrated the benefits of algorithms-by-blocks, coupled with the SuperMatrix run-time system, for all three major (dense and banded) factorization operations: the Cholesky factorization [6, 22], the QR factorization [20], and now the LU factorization. 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.

For the particular operation studied in this paper, the LU factorization of a dense matrix, we have shown that an algorithm-by-blocks first developed for out-of-core computation can be converted to a parallel algorithm that targets

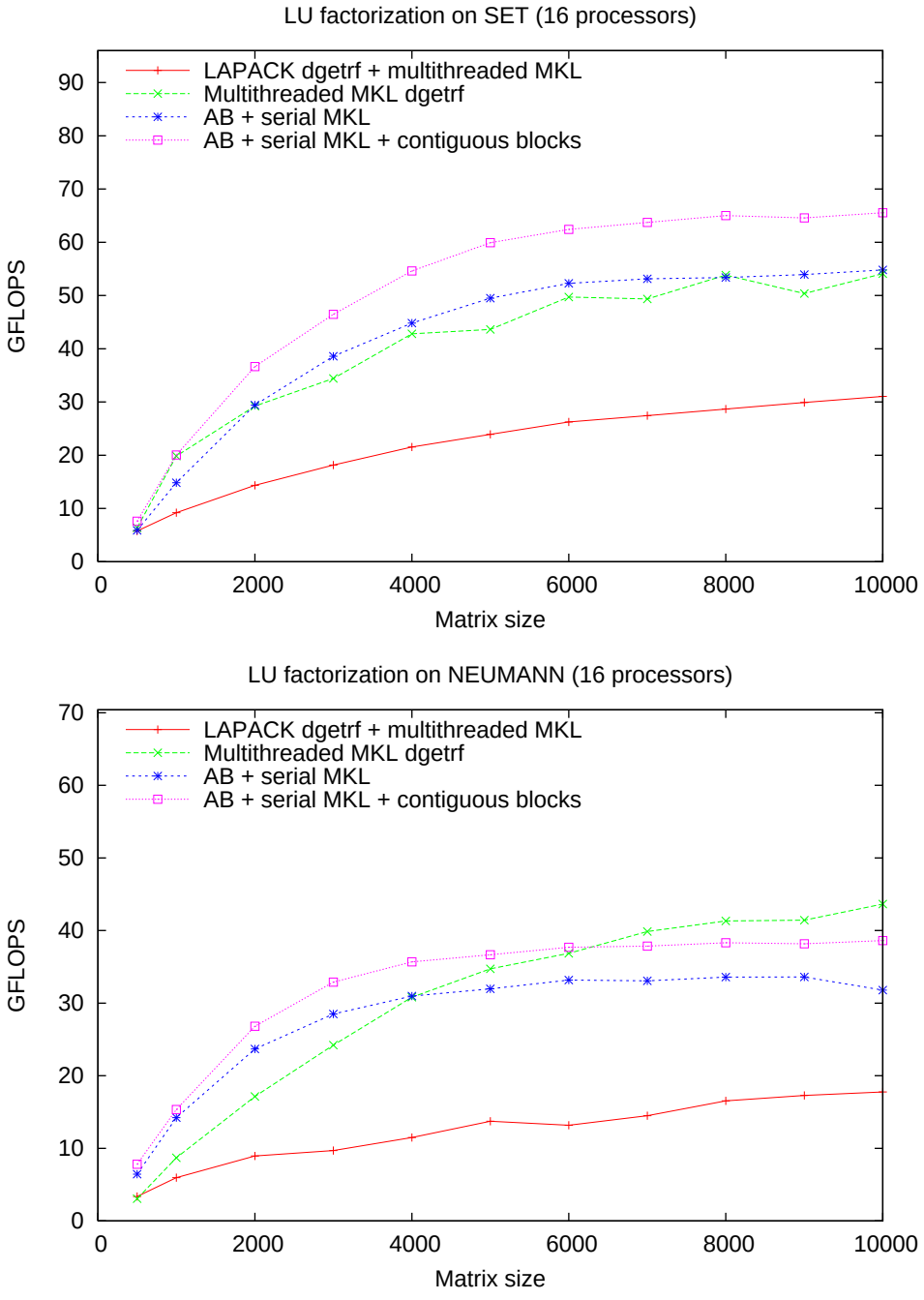


**Fig. 5.** Performance of the LU factorization algorithms.

multithreaded architectures. By executing the algorithm with the SuperMatrix run-time system on matrices stored by blocks, remarkable performance was attained relative to the LAPACK implementations of routine `dgetrf`.

**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).*

We thank John Gilbert and Vikram Aggarwal from the University of California at Santa Barbara for granting the access to the NEUMANN platform.

## References

1. 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.
2. 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.
3. 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.
4. 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.
5. 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.
6. 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.
7. 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.
8. 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.
9. 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.

10. 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.
11. Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, 3rd edition, 1996.
12. 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.
13. 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.
14. J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufman, 3rd edition, 2003.
15. 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.
16. 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.
17. 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.
18. 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.
19. 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.
20. 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.
21. Gregorio Quintana-Ortí, Enrique S. Quintana-Ortí, Ernie Chan, Robert van de Geijn, and Field G. Van Zee. Design and scheduling of an algorithm-by-blocks for the LU factorization on multithreaded architectures. FLAME Working Note #26 TR-07-50, The University of Texas at Austin, Department of Computer Sciences, September 2007.
22. 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.
23. G. W. Stewart. *Introduction to Matrix Computations*. Academic Press, Orlando, Florida, 1973.