An Algorithm-by-Blocks for SuperMatrix Band Cholesky Factorization

Gregorio Quintana-Ortí[1], Enrique S. Quintana-Ortí[1], Alfredo Remón[1], and Robert A. van de Geijn[2]

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

Abstract. We pursue the scalable parallel implementation of the factorization of band matrices with medium to large bandwidth targeting SMP and multi-core architectures. Our approach decomposes the computation into a large number of fine-grained operations exposing a higher degree of parallelism. The SuperMatrix run-time system allows an out-of-order scheduling of operations that is transparent to the programmer. Experimental results for the Cholesky factorization of band matrices on two parallel platforms with sixteen processors demonstrate the scalability of the solution.

Key words: Cholesky factorization, band matrices, high-performance, dynamic scheduling, out-of-order execution, linear algebra libraries.

1 Introduction

How to extract parallelism from linear algebra libraries is being reevaluated with the emergence of SMP architectures with many processors, multi-core systems that will soon have many cores, and hardware accelerators such as the Cell BE processor or graphics processors (GPUs). In this note, we demonstrate how techniques that have shown to be extremely useful for the parallelization of dense factorizations in this context [7, 8, 17, 18, 6, 5] can also be extended for the factorization of band matrices [19]. The result is an algorithm-by-blocks that yields high performance and scalability for matrices of moderate to large bandwidth while keeping the implementation simple by various programmability measures. To illustrate our case, we employ the Cholesky factorization of band symmetric positive definite matrices as a prototypical example. However, the same ideas apply to algorithms for the LU and QR factorization of band matrices.

The contributions of this paper include the following:

– We demonstrate that high performance can be attained by programs coded at a high level of abstraction, even by algorithms for complex operations like the factorization of band matrices and on sophisticated environments like many-threaded architectures.

– We show how the SuperMatrix run-time system supports out-of-order computation on blocks transparent to the programmer leading to a solution which exhibits superior scalability for band matrices.
– We also show how the FLASH extension of FLAME supports storage by blocks for band matrices different from the commonly-used packed storage used in LAPACK [1].
– We compare and contrast a traditional blocked algorithm for the band Cholesky factorization to a new algorithm-by-blocks.

This paper is structured as follows. In Section 2 we describe a blocked algorithm for the Cholesky factorization of a band matrix which reflects the state-of-the-art for this operation. Then, in Section 3, we present an algorithm-by-blocks which advances operations that are in the critical path from "future" iterations. The FLAME tools employed to implement this algorithm are outlined in Section 4. In Section 5 we demonstrate the scalability of this solution on a CC-NUMA with sixteen Intel Itanium2 processors and an SMP with 8 AMD Opteron (dual core) processors. Finally, in Section 6 we provide a few concluding remarks.

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

2 Computing the Cholesky factorization of a band matrix

Given a symmetric positive definite matrix A of dimension $n \times n$, its Cholesky factorization is given by $A = LL^T$, where L is an $n \times n$ lower triangular matrix. (Alternatively, A can be decomposed into the product $A = U^T U$ with U an $n \times n$ upper triangular matrix, a case that we do not pursue further.) In case A presents a band structure with upper and lower bandwidth k_d (that is, all entries below the $k_d + 1$ subdiagonal and above the $k_d + 1$ superdiagonal are zero), then L presents the same lower bandwidth as A. Exploiting the band structure of A when $k_d \ll n$ leads to important savings in both storage and computation. This was already recognized in LINPACK and later in LAPACK which includes unblocked and blocked routines for the Cholesky factorization of a band matrix.

2.1 The blocked algorithm in routine PBTRF

It is well-known that high performance can be achieved in a portable fashion by casting algorithms in terms of matrix-matrix multiplication [1, 11]. Figure 1 illustrates how the LAPACK blocked routine PBTRF does so for the Cholesky factorization of a band matrix with non-negligible bandwidth. For simplicity we consider there and hereafter that n and k_d are exact multiples of k_d and n_b,

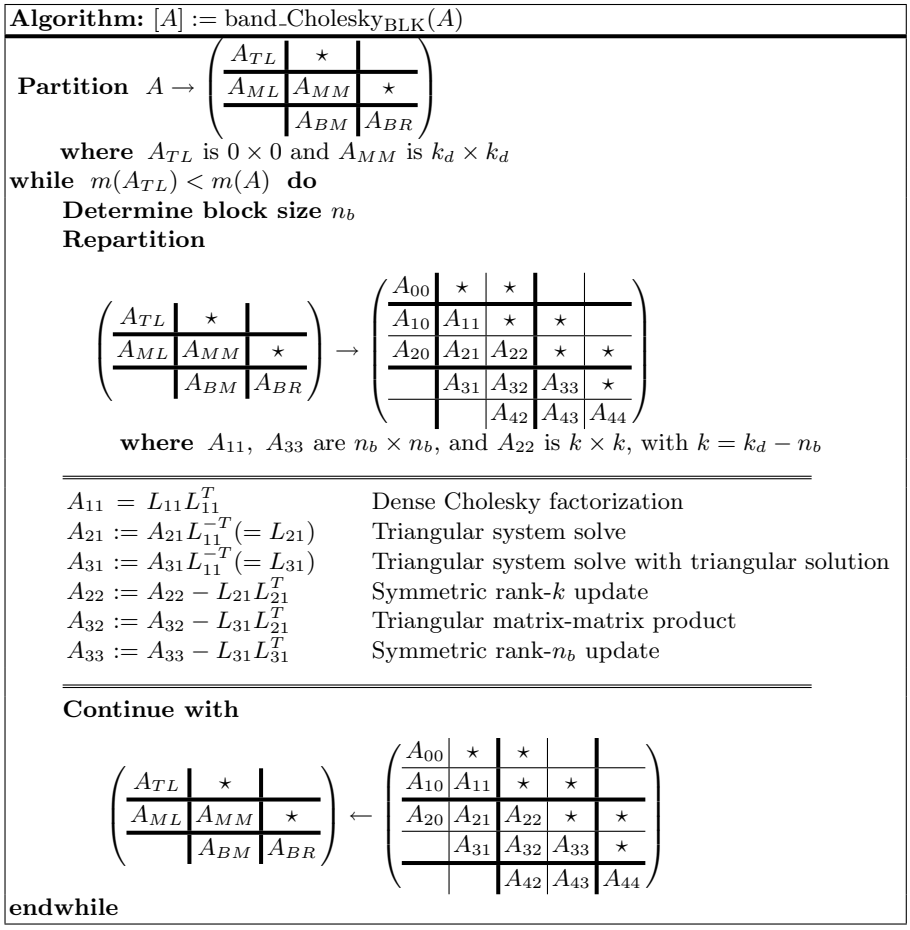

Fig. 1. Blocked algorithm for the Cholesky factorization of a band matrix.

respectively. Provided $n_b \ll k_d$ most of the computations in the algorithm are cast into the symmetric rank-k update of A_{22}.

Upon completion of the factorization using the algorithm in the figure, the elements of L overwrite the corresponding entries of A. The "$\star$" symbols in the figure denote symmetric parts in the upper triangle of A which are not accessed/referenced.

2.2 Packed storage in routine PBTRF

Routine PBTRF employs a packed format to save storage. Specifically, the symmetry of A requires only its lower (or upper) triangular part to be stored, which is saved following the pattern illustrated in Figure 2 (right). As they are com-

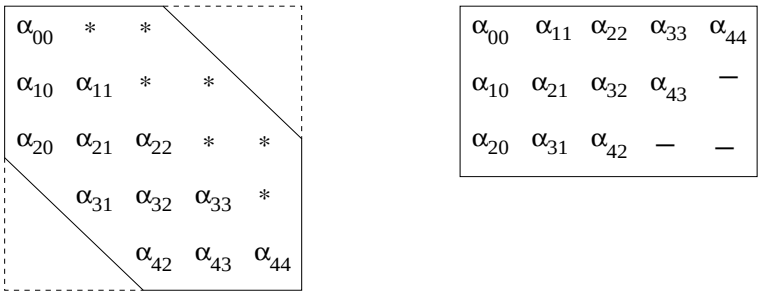

Fig. 2. Symmetric 5×5 band matrix with bandwidth $k_d = 2$ (left) and packed storage used in LAPACK (right). The '$*$' symbols denote the symmetric entries of the matrix which are not stored in the packed format.

puted, the elements of L overwrite the corresponding entries of A in the packed matrix.

Due to A_{31}/L_{31} having only their upper triangular parts stored, some operations in the actual implementation of the algorithm in Figure 1 need special care, as described next. In particular, in order to solve the triangular linear system for A_{31}, a copy of the upper triangular part of A_{31} is first obtained in an auxiliary workspace W of dimension $n_b \times n_b$ with its subdiagonal entries set to zero; the BLAS-3 solver TRSM is then used to obtain $W := WL_{11}^{-T}$ $(= L_{31})$. Next, the update of A_{32} is computed as a general matrix product using BLAS-3 kernel GEMM to yield $A_{32} := A_{32} - WL_{21}^T$. Finally, the update of A_{33} is obtained using BLAS-3 kernel SYRK as $A_{33} := A_{33} - WW^T$, and the upper triangular part of W is written back to A_{31}.

2.3 Parallelism within the BLAS

Blocked implementations of the band Cholesky factorization are typically written so that the bulk of the computation is performed by calls to the *Basic Linear Algebra Subprograms* (BLAS), a standardized interface to routines that carry out operations as matrix-vector (level-2 BLAS) and matrix-matrix multiplication (level-3 BLAS).

Parallelism can be attained within each call to a BLAS routine with the following benefits:

- The approach allows legacy libraries, such as LAPACK, to be used without change.
- Parallelism within suboperations, e.g., the update of A_{11}–A_{33} in Figure 1, can be exploited through multithreaded implementations of BLAS. However, note that in practice $k_d \gg n_b$ and n_b is typically small so that the major bulk of the computations is in the update of A_{22} while the remaining operations may be too small to gain any benefit from the use of a multithreaded implementation of BLAS.

Disadvantages, on the other hand, include:

- The parallelism achieved is only as good as the underlying multithreaded implementation of the BLAS.
- The end of each call to a BLAS operation becomes a synchronization point (a barrier) for the threads. In [20] it is shown how the updates of A_{21}, A_{31} can be merged into a single triangular linear system solve and the updates A_{22}, A_{32}, and A_{33} into a single symmetric rank-k_d update, so that a coarser grain of parallelism is obtained and the number of synchronization points is diminished. The performance increase which can be gained from this approach is modest, within 5–10% depending on the bandwidth of the matrix and the architecture.
- For many operations the choice of algorithmic variant can severely impact the performance that is achieved.

In the next section we propose an algorithm composed of operations with finer granularity to overcome these difficulties.

3 An Algorithm-by-blocks

Since the early 1990s, various researchers [10, 12, 13, 16] have proposed that matrices should be stored by blocks as opposed to the more customary column-major storage used in Fortran and row-major storage used in C. Doing so recursively is a generalization of that idea. The original reason was that by storing matrices contiguously a performance benefit would result. More recently, we have proposed that the blocks should be viewed as units of data and operations with blocks as units of computation [7, 9]. In the following we show how to decompose the updates in the algorithm for the band Cholesky factorization so that an algorithm-by-blocks results which performs all operations on "tiny" $n_b \times n_b$ blocks.

For our discussion below, assume $k = pn_b$ for the blocked algorithm in Figure 1. Then, given the dimensions imposed by the partitionings on A,

$$\left(\begin{array}{c|c|c} A_{11} & \star & \star \\ \hline A_{21} & A_{22} & \star \\ \hline A_{31} & A_{32} & A_{33} \end{array}\right) \rightarrow \left(\begin{array}{c|cccc|c} A_{11} & \star & \star & \star & \star & \star \\ \hline A_{21}^0 & A_{22}^{00} & \star & \star & \star & \star \\ A_{21}^1 & A_{22}^{10} & A_{22}^{11} & \star & \star & \star \\ \vdots & \vdots & \vdots & \ddots & \star & \star \\ A_{21}^{p-1} & A_{22}^{p-1,0} & A_{22}^{p-1,1} & \cdots & A_{22}^{p-1,p-1} & \star \\ \hline A_{31} & A_{32}^0 & A_{32}^1 & \cdots & A_{31}^{p-1} & A_{33} \end{array}\right),$$

where all blocks are $n_b \times n_b$. Therefore, the update $A_{21} := A_{21}L_{11}^{-T}$ in the blocked algorithm can be decomposed into

$$\left(\begin{array}{c} A_{21}^0 \\ A_{21}^1 \\ \vdots \\ A_{21}^{p-1} \end{array}\right) := \left(\begin{array}{c} A_{21}^0 \\ A_{21}^1 \\ \vdots \\ A_{21}^{p-1} \end{array}\right) L_{11}^{-T}, \tag{1}$$

which corresponds to p triangular linear systems on $n_b \times n_b$ blocks. Similarly, the update $A_{22} := A_{22} - L_{21}L_{21}^T$ becomes

$$\begin{pmatrix} A_{22}^{00} & \star & \star & \star \\ A_{22}^{10} & A_{22}^{11} & \star & \star \\ \vdots & \vdots & \ddots & \star \\ A_{22}^{p-1,0} & A_{22}^{p-1,1} & \cdots & A_{22}^{p-1,p-1} \end{pmatrix} := \begin{pmatrix} A_{22}^{00} & \star & \star & \star \\ A_{22}^{10} & A_{22}^{11} & \star & \star \\ \vdots & \vdots & \ddots & \star \\ A_{22}^{p-1,0} & A_{22}^{p-1,1} & \cdots & A_{22}^{p-1,p-1} \end{pmatrix} - \begin{pmatrix} A_{21}^0 \\ A_{21}^1 \\ \vdots \\ A_{21}^{p-1} \end{pmatrix} \begin{pmatrix} A_{21}^0 \\ A_{21}^1 \\ \vdots \\ A_{21}^{p-1} \end{pmatrix}^T, \tag{2}$$

where we can identify p symmetric rank-n_b updates (for the $n_b \times n_b$ diagonal blocks) and $(p^2/2 - p/2)$ general matrix products (for the $n_b \times n_b$ subdiagonal blocks). Finally, the update $A_{32} := A_{32} - L_{31}L_{21}^T$ is equivalent to

$$\left(A_{32}^0 \; A_{32}^1 \; \ldots \; A_{31}^{p-1} \right) := \left(A_{32}^0 \; A_{32}^1 \; \ldots \; A_{31}^{p-1} \right) - A_{31} \begin{pmatrix} A_{21}^0 \\ A_{21}^1 \\ \vdots \\ A_{21}^{p-1} \end{pmatrix}^T \tag{3}$$

which, given the upper triangular structure of A_{31}, corresponds to p triangular matrix-matrix products of dimension $n_b \times n_b$.

The first key point to realize here is that all the operations on blocks in (1) are independent and therefore can be performed concurrently. The same holds for the operations in (2) and also for those in (3). By decomposing the updates of A_{21}, A_{22}, and A_{32} as in (1)–(3) more parallelism is exposed at the block level in the algorithm in Figure 1.

The second key point is that some of the block operations in (1) can proceed in parallel with block operations in (2) and (3). Thus, for example, $A_{21}^1 := A_{21}^1 L_{11}^{-1}$ is independent from $A_{22}^{00} := A_{22}^{00} - A_{21}^0 (A_{21}^0)^T$ and both can be computed in parallel. This is a fundamental difference compared with a parallelization entirely based on a parallel (multithreaded) BLAS, where each BLAS call is a synchronization point so that, e.g., no thread can be updating (a block of) A_{22} before the update of (all blocks within) A_{21} is completed.

4 The FLAME tools

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

4.1 FLAME

FLAME is a methodology for deriving and implementing dense linear algebra operations [3]. The (semiautomatic) application of this methodology produces

provably correct algorithms for a wide variety of linear algebra computations. The use of the Application Programming Interface (API) for the C programming language allows an easy translation of FLAME algorithm to C code, as illustrated for dense linear algebra operations in [4].

4.2 FLASH

One naturally thinks of matrices stored by blocks as matrices of matrices. As a result, if the API encapsulates information that describes a matrix in an object, as FLAME 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. Multiple layers of this idea can be used if multiple hierarchical layers in the matrix are to be exposed. We call this extension to the FLAME API the FLASH API [15]. Examples of how simpler operations can be transformed from FLAME to FLASH implementations can be found in [7, 9].

The FLASH API provides a manner to store band matrices that is conceptually different from that of LAPACK. Using the FLASH API, a blocked storage is easy to implement where only those $(n_b \times n_b)$ blocks with elements within the (nonzero) band are actually stored. The result is a packed storage which roughly requires same the order of elements as the traditional packed scheme but which decouples the logical and the physical storage patterns, yielding higher performance. Special storage schemes for triangular and symmetric matrices can still be combined for performance or to save space within the $n_b \times n_b$ blocks.

4.3 SuperMatrix

Given a FLAME algorithm implemented in code using the FLAME/C interface, the SuperMatrix run-time system first builds a Directed Acyclic Graph (DAG) that represents all operations that need to be performed together with the dependencies among these. The run-time system then uses the information in the DAG to schedule operations for execution dynamically, as dependencies are fulfilled. These two phases, construction of the DAG and scheduling of operations, can proceed completely transparent to the specific implementation of the library routine. For further details on SuperMatrix, see [7, 9].

We used OpenMP to provide multithreading facilities where each thread executes asynchronously. We have also implemented SuperMatrix using the POSIX threads API to reach a broader range of platforms.

Approaches similar to SuperMatrix have been described for more general irregular problems in the frame of the Cilk project [14] (for problems that can be easily formulated as divide-and-conquer, unlike the band Cholesky factorization), and for general problems also but with the specific target of the Cell processor in the CellSs project [2].

5 Experiments

In this section, we evaluate two implementations for the Cholesky factorization of a band matrix with varying dimension and bandwidth. 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 (96×10^9 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 two parallelizations of the Cholesky factorization:

- **LAPACK** DPBTRF **+ multithreaded MKL**. LAPACK 3.0 routine DPBTRF linked to multithreaded BLAS in MKL.
- **AB + serial MKL**. Our implementation of the algorithm-by-blocks linked to serial BLAS in MKL.

When hand-tuning block sizes, a best-effort was made to determine the best values of n_b in both cases.

Figures 3 and 4 report the performance of the two parallel implementations for band matrices of order $n = 2000$ and $n = 5000$ (from bigger matrices higher performance can be expected due to the increment in parallelism) with varying dimension of the bandwidth and number of processors. The first thing to note from this experiment is the lack of scalability of the solution based on a multithreaded BLAS (plots on the left column): as more processors are added to the experiment, the left plots in the figure shows a notable drop in the performance so that using more than 2 or 4 processors basically yields no gain or even results in a performance decrease. The situation is different for the algorithm-by-blocks (plots on the right-hand side): For example, while using 4 or 8 processors on SET for a matrix of bandwidth below 200 attains a similar GFLOPS rate, using 8 processors for matrices of larger bandwidth achieves a significant performance increase. A similar behavior occurs when all 16 processors of SET are employed but at a higher threshold, $k_d \approx 450$.

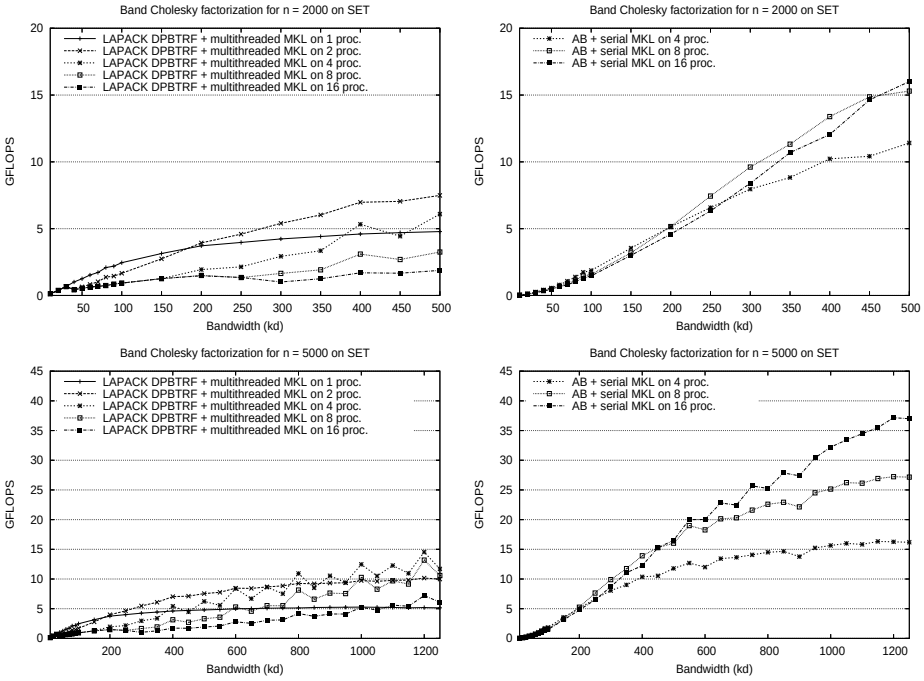

Fig. 3. Performance of the band Cholesky factorization algorithms on 1, 2, 4, 8, and 16 CPUs of SET.

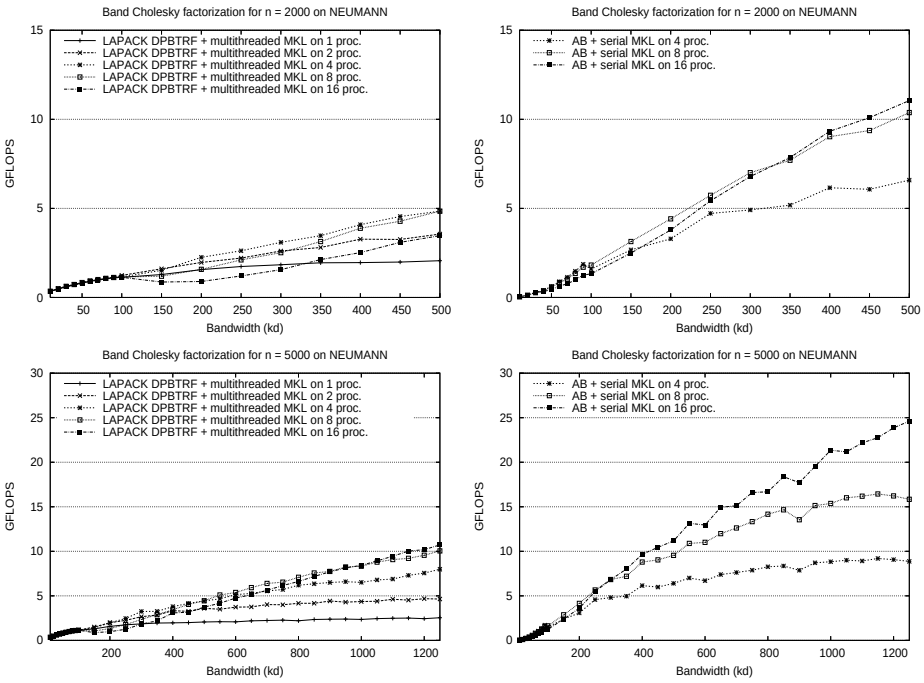

Fig. 4. Performance of the band Cholesky factorization algorithms on 1, 2, 4, 8, and 16 CPUs of NEUMMAN.

Figure 5 compares the two parallel implementations using the optimal number of processors: 2 (n=2000) and 4 (n=5000) on SET for the LAPACK DPBTRF+ multithreaded MKL implementation; 4 (n=2000) and 16 (n=5000) for this same algorithm on NEUMANN; and 16 for the AB + serial MKL implementation on both platforms. From this experiment it is clear the benefits of using an algorithm-by-blocks on a machine with a large number of processors.

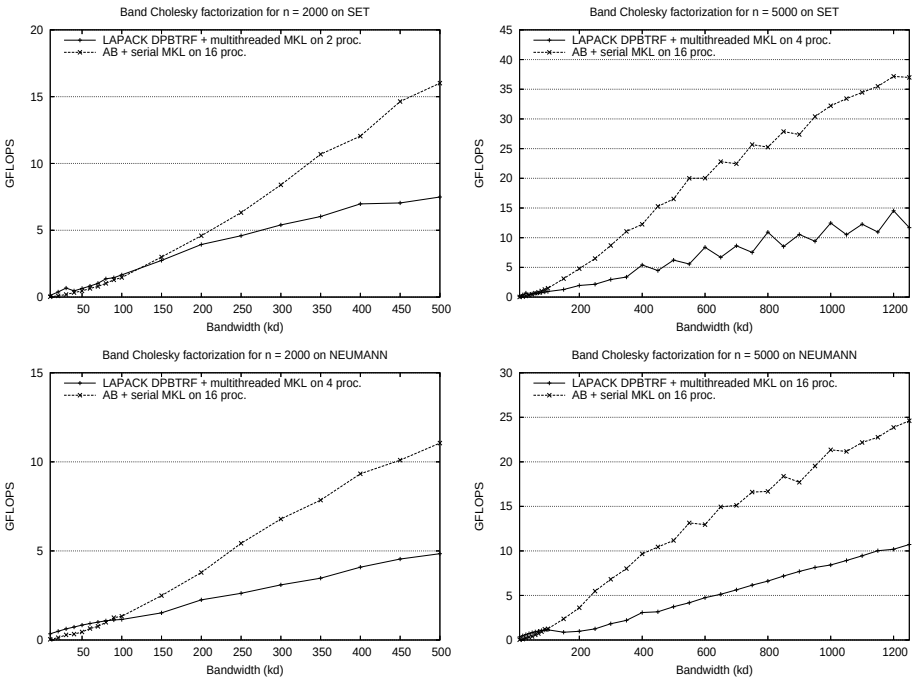

Fig. 5. Performance of the band Cholesky factorization algorithms.

6 Conclusions

We have presented an extension of SuperMatrix that yields algorithms-by-blocks for the Cholesky, LU (with and without pivoting) and QR factorizations of band matrices. The programming effort was greatly reduced by coding the algorithms with the FLAME/C and FLASH APIs. Using the algorithm-by-blocks, the SuperMatrix run-time system generates a DAG of operations which is then used to schedule out-of-order computation on blocks transparent to the programmer.

The results on two different parallel architectures for an algorithm-by-blocks for the band Cholesky factorization of matrices with medium to large band-

width clearly report higher performance and superior scalability to those of a traditional multithreaded approach using LAPACK.

Acknowledgments

We thank the other members of the FLAME team for their support. This research was partially sponsored by NSF grants CCF–0540926 and CCF–0702714. Gregorio Quintana-Ortí, Enrique S. Quintana-Ortí, and Alfredo Remón were supported by by the CICYT project TIN2005-09037-C02-02 and FEDER. This work was partially carried out when Alfredo Remn was visiting the Chemnitz University of Technology with a grant from the programme *Plan 2007 de Promocin de la Investigacin* of the *Universidad Jaime I.*

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

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).

References

1. E. Anderson, Z. Bai, J. Demmel, J. E. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. E. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, 1992.
2. Pieter Bellens, Josep M. Perez, Rosa M. Badia, and Jesus 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, Gregorio Quintana-Ortí Enrique S. Quintana-Ortí, and Robert van de Geijn. SuperMatrix: A multithreaded runtime scheduling system for algorithms-by-blocks. In *PPoPP'08: Proceedings of the 13th ACM SIGPLAN symposium on Principles and practices of parallel programming*, New York, NY, USA, 2008. ACM Press. To appear.

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. Jack J. Dongarra, Iain S. Duff, Danny C. Sorensen, and Henk A. van der Vorst. *Solving Linear Systems on Vector and Shared Memory Computers*. SIAM, Philadelphia, PA, 1991.
12. 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.
13. Greg Henry. BLAS based on block data structures. Theory Center Technical Report CTC92TR89, Cornell University, Feb. 1992.
14. C. Leiserson and A. Plaat. Programming parallel applications in Cilk. *SINEWS: SIAM News*, 1998.
15. 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.
16. 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.
17. 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. In *16th Euromicro Int. Conference on Parallel, Distributed and Network-based Processing*. IEEE, 2008. To appear.
18. 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.
19. 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.
20. Alfredo Remón, Enrique S. Quintana-Ortí, and Gregorio Quintana-Ortí. Cholesky factorization of band matrices using multithreaded BLAS. In *PARA'06*, Lecture Notes in Computer Science. Springer-Verlag, 2007. To appear.