SIAM REVIEW
Vol. 46, No. 1, pp. 3–45

Recursive Blocked Algorithms and Hybrid Data Structures for Dense Matrix Library Software*

Erik Elmroth†
Fred Gustavson‡
Isak Jonsson†
Bo Kågström†

Abstract. Matrix computations are both fundamental and ubiquitous in computational science and its vast application areas. Along with the development of more advanced computer systems with complex memory hierarchies, there is a continuing demand for new algorithms and library software that efficiently utilize and adapt to new architecture features. This article reviews and details some of the recent advances made by applying the paradigm of recursion to dense matrix computations on today's memory-tiered computer systems. Recursion allows for efficient utilization of a memory hierarchy and generalizes existing fixed blocking by introducing automatic variable blocking that has the potential of matching every level of a deep memory hierarchy. Novel recursive blocked algorithms offer new ways to compute factorizations such as Cholesky and QR and to solve matrix equations. In fact, the whole gamut of existing dense linear algebra factorization is beginning to be reexamined in view of the recursive paradigm. Use of recursion has led to using new hybrid data structures and optimized superscalar kernels. The results we survey include new algorithms and library software implementations for level 3 kernels, matrix factorizations, and the solution of general systems of linear equations and several common matrix equations. The software implementations we survey are robust and show impressive performance on today's high performance computing systems.

Key words. recursion, automatic variable blocking, superscalar, GEMM-based, level 3 BLAS, hybrid data structures, superscalar kernels, SMP parallelization, library software, LAPACK, SLICOT, ESSL, RECSY, dense linear algebra, factorizations, matrix equations

AMS subject classifications. 15-02, 15A23, 15A24, 65F05, 65F15, 65F20, 65F35, 68M20, 68W10

DOI. 10.1137/S0036144503428693

1. Introduction.

1.1. The Significance of Linear Algebra Software.

One of the first uses of digital computers, more than 50 years ago, was to solve linear systems, and linear algebra software has played a central role in computing ever since. The preface of the pioneering 1971 Wilkinson and Reinsch book of Algol codes [101] notes that "...algorithms

*Received by the editors May 23, 2003; accepted for publication (in revised form) January 9, 2004; published electronically February 2, 2004. This work was supported by the Swedish Research Council under grants TFR 98-604 and VR 621-2001-3284, and by the Swedish Foundation for Strategic Research (SSF) under grant A3 02:128.

http://www.siam.org/journals/sirev/46-1/42869.html

†Department of Computing Science and HPC2N, Umeå University, S-901 87 Umeå, Sweden (elmroth@cs.umu.se, isak@cs.umu.se, bokg@cs.umu.se).

‡IBM T. J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598 (fg2@us.ibm.com).

[of linear algebra] are perhaps the most widely used in numerical analysis . . . "; it also observes, with great prescience, that ". . . preparation of a fully tested set of algorithms is a far greater task than had been anticipated."

The early work of many researchers (see, e.g., [31, 39, 92]) established the robustness and stability of fundamental matrix techniques, including Cholesky, LU, and QR factorizations and related algorithms. The next step was the time-consuming production of high-quality linear algebra software libraries such as LINPACK [13] and EISPACK [91]. By any measure, these libraries and their descendants have been extremely successful. In addition to their presence at the heart of MATLAB [76], LINPACK and EISPACK have each been requested more than three million times, and LAPACK ([4]; see section 1.2) has been requested more than twenty million times, from the online software repository NETLIB (see www.netlib.org/master_counts2.html).

The prevalence of linear algebraic computations in scientific, engineering, medical, and business applications remains striking today. The worldwide publicity in 2002 about the Japanese Earth Simulator, as of 2004 still the world's fastest computer, cited its achievement of 35.6 teraflops (35.6×10^{12} floating-point operations per second) in solving a dense square linear system $Ax = b$ of size 1,041,216 using the LINPACK benchmark [96]. Matrix computations are ubiquitous—for instance, in modeling and simulation of problems ranging from galaxies to the nanoscale, and in real-time airline scheduling and medical imaging.

Although the importance of linear algebra is evident, it might appear that there is very little new to be said about linear algebra software, especially given the relative mathematical simplicity of standard algorithms such as Gaussian elimination. However, this is not at all the case, largely because of continuing advances in computer hardware that dramatically affect the performance of even the most basic linear algebraic computations. As long ago as 1976, it was observed that decisions made by the compiler could degrade the speed of a "simple" triangular solve [78]. As we shall discuss, maximum performance on high-end computers can be achieved only if careful attention is paid to the algorithm as well as to the hardware and the compiler. An additional complication is that algorithmic adaptations should be, as much as possible, automatic, rather than requiring *ab initio* implementation for every new architecture.

Many complexities arise from nonobvious and often unpredictable interactions between the compute engine (the processors) and the data upon which they perform calculations. Today's computers have extremely fast processors, but memory access speeds are relatively slow. Thus the performance of algorithms is frequently limited by the need to move large amounts of data between memory and the processor. This problem is particularly acute in dense matrix computations where algorithms such as factorization require repeatedly sweeping through all elements of the matrix. As a result, features of computer hardware have profoundly influenced the implementation as well as the abstract description of matrix algorithms.

1.2. A Short History of Linear Algebra Software and Computer Hardware. To retain portability and efficiency across diverse and changing platforms, in the late 1970s the major developers of linear algebra software made a crucial decision: to express matrix algorithms in terms of basic operations of specified functionality. This concept—usually called the BLAS (Basic Linear Algebra Subprograms) [74]—allows definition of machine-independent algorithms at the highest level of abstraction as well as inclusion of efficient code, tailored to the hardware, in the inner loops of any particular implementation. The earliest BLAS, called "level 1," involve vector-scalar and vector-vector operations, such as dot product and "AXPY" ($y \leftarrow y + \alpha x$, where

x and y are vectors and α is a scalar). On a computer of this era, the time needed to access any element in memory was more or less the same, which meant that the operation (flop) count was an accurate estimate of performance.

Through the 1980s and into the early 1990s, the most powerful computers, such as those built by Cray Research, were based on vector architectures in which the same operation is simultaneously performed on every element of a vector. This structure was exploited in linear algebra software by emphasizing matrix-vector operations, called "level 2" BLAS, such as "GEMV" ($y \leftarrow \beta y + \alpha Ax$, where A is a matrix and α and β are scalars) [23].

A second, deeply influential development in computer architecture was *cache memory*, a small fast memory holding recently accessed data. Cache memory is a great benefit for calculations characterized by *locality*, in which many (fast) computations are performed on the same data, thereby avoiding (slow) data accesses. With cache memory, speed depends on having the needed data in the cache; if a program requires data not in the cache, an undesirable *cache miss* occurs. During the late 1980s and into the 1990s, architectural complexity increased as *hierarchical memory systems* were introduced, featuring multiple levels of cache storage with varying sizes and access speeds.

Obtaining good performance with such systems required recasting linear algebra algorithms as much as possible in terms of operations on *blocks* (submatrices), thereby reducing the probability of cache misses. (More generally, "blocking" refers to creating groups of related data elements with the property that calculations are performed on the entire group.) The linear algebra community established a forum to define the associated "level 3" BLAS [22, 21], which specify matrix-matrix operations, such as "GEMM" (general matrix multiply and add),

$$C \leftarrow \beta C + \alpha AB, \tag{1.1}$$

where A, B, and C are (sub)matrices; e.g., see section 3 in [36]. The algorithms in LINPACK and EISPACK were then rewritten with level 1, 2, and 3 BLAS, assuming that computer manufacturers would produce their own high-performance BLAS. The resulting LAPACK library [4] provided new functionality and also included new and improved algorithms such as divide-and-conquer methods for symmetric eigenvalue computations [17, 4]. Like its predecessors, LAPACK is based on a modular design intended to produce portability, robustness, and good performance through use of the BLAS (see, e.g., [25]).

But this is not the end of the story. In the last decade memory hierarchies have become much deeper (i.e., they contain more levels with more varied properties); consequently the overall performance of LAPACK and the level 3 BLAS has tended to degrade relative to peak calculation speed. And this brings us to one of our main themes: recursive blocked algorithms.

1.3. Hierarchical Blocking: Motivation and Implications. As already observed, a key to efficient matrix computations on hierarchical memory machines is *blocking*, or grouping of the matrix elements. We shall show later that significant speedups can result from new blocking strategies, but first we present some general principles and an introductory discussion of blocking.

1.3.1. Two Principles. The linear algebra software considered in this paper relies on the following two ideas.

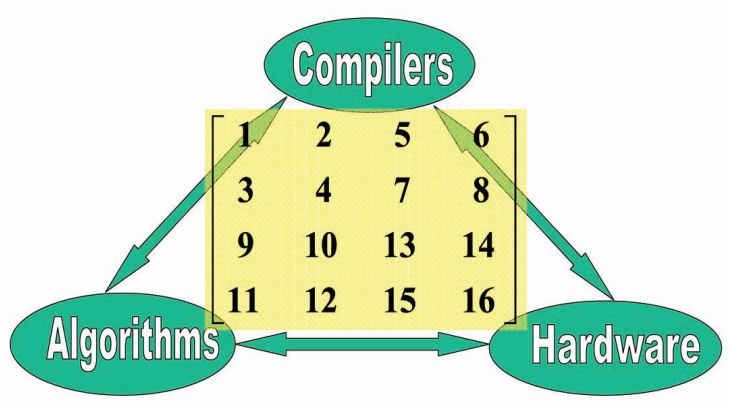

Fig. 1.1 *The fundamental AHC triangle overlaid with a matrix of blocks in recursive block row (Z-Morton) ordering* [46, 102].

Recursion. Recursion, exemplified by calculation of $n!$, is a fundamental concept for expressing algorithms in computer science. Although recursive algorithm descriptions are typically more elegant and concise than those couched in iterative and conditional loops, software based on recursion was condemned for many years because compilers could not deal efficiently with the overhead of recursive function calls. However, this situation is changing; modern compilers can handle recursion, and today's programming languages (even Fortran!) allow recursive subprograms. A central theme of this paper is the value of recursion as a general technique for producing dense linear algebra software.

The algorithms-hardware-compilers triangle. As mentioned in section 1.1, it has been observed for a long time that performance of an implemented algorithm depends not only on the computational steps but also on the compiler and hardware; some related work is reviewed in section 7. We find it helpful to think of the "fundamental triangle" shown in Figure 1.1, which conveys the tight connections needed among algorithms, hardware, and compilers. Formal recognition of the elements in Figure 1.1 was an essential ingredient in the "algorithms and architecture approach" [1], which matches algorithms to architecture, and vice versa. This approach was used to produce software in the Engineering and Scientific Subroutine Library (ESSL) [56] and the GEMM-based level 3 BLAS project [66, 67]. In the latter, all level 3 BLAS operations were rewritten as GEMM operations (see (1.1)) combined with level 2 BLAS, so that only a highly optimized GEMM kernel was needed to produce all other level 3 BLAS "for free."

The algorithms and architecture approach was the precursor of much of the authors' work on recursion and new data structures for linear algebra software. The same approach is, of course, a principal guideline in compiler technology. The idea of blocking features in the influential 1991 papers by Wolf and Lam [103] and Lam, Rothberg, and Wolf [73]; see the discussion in section 7. Some of the results described in this paper have already had impact on the compiler community, e.g., regarding compiler blockability [14] and automatic generation of recursive blocked codes [2, 104].

1.3.2. Why Blocking Matters. The importance of blocking in high-performance linear algebra software follows from the algorithms-compilers-hardware triangle of Figure 1.1: matrix elements should be grouped to match the structure of a hierarchical memory machine, which we now sketch.

At the top of a computer memory hierarchy are the registers where all computations (floating point, integer, and logical) take place; these have the shortest access times but the smallest storage capacities. Between the registers and main memory, there are one or more levels of cache memory with slower access time and increasing size. Closest to the registers is the first-level cache memory (L1 cache), the fastest but smallest; next are the second-level (L2) and possibly third-level (L3) cache memories. Below main memory are levels of secondary storage with larger capacity but slower access speed, such as disk and tape.

Different levels in the memory hierarchy display vastly different access times. For example, register and L1 cache access speeds are typically on the order of nanoseconds, whereas disk access speeds are in milliseconds—a difference of 10^6. Furthermore, access time changes by a factor of five or ten between each level. Thus the ideal for such a system is to perform as many calculations as possible on data that resides in the fastest cache. The storage capacity of each level also varies greatly. Registers may hold up to a few kilobytes of data, L1 cache up to a few hundred kilobytes, the highest-level cache up to a few megabytes (10^6 bytes), while today's main memory can hold several gigabytes (10^9 bytes).

1.3.3. Blocking Strategies. Various strategies are known for blocking data in order to exploit a hierarchical memory structure. The classical way is *explicit multilevel blocking*, where each index/loop set matches a specific level of the memory hierarchy. This requires a detailed knowledge of the architecture and (usually) a separate blocking parameter for each level. *Register blocking* and *cache blocking* refer to variants of this idea designed for efficient reuse of, respectively, data in the registers and one or more levels of cache.

The memory of most computers is laid out in blocks of fixed size, called *pages*. At any instant of computer time there is a set of "fast" pages, sometimes called the working set, which reside in the *translation look-aside buffer* (TLB). The term *TLB blocking* means a strategy designed so that memory is mostly accessed in the working set.

In contrast to these approaches, *recursive blocking*, which combines recursion and blocking, leads to an automatic variable blocking with the potential for matching the memory hierarchies of today's high-performance computing systems. Recursive blocking means that "traditional" recursion is to terminate when the size of the controlling recursion blocking parameter becomes smaller than some predefined value. This assures that all leaf computations in the recursion tree will usually be a substantial level 3 (matrix-matrix) computation. Let us illustrate with a simple matrix addition $C = A+B$, where all matrices are $n \times n$, n is an even number, and the blocking parameter is $n/2$. By first splitting the matrix add operation in the row dimension and then splitting in the column dimension, we get four leaf subtasks (red nodes in Figure 1.2) each corresponding to a level 3 operation $C_{ij} = A_{ij} + B_{ij}$ with all square submatrices (or blocks) of size $n/2 \times n/2$. In this case, the four leaf subtasks can be performed independently, e.g., using different processors on a shared memory computer system, and the global result is obtained simultaneously since this problem involves only a "divide" phase. But as we will see for more complicated and nonregular matrix operations, we may have to "glue" results from all nodes, including leaf, together using some additional computations. Different problems lead to different dependencies between

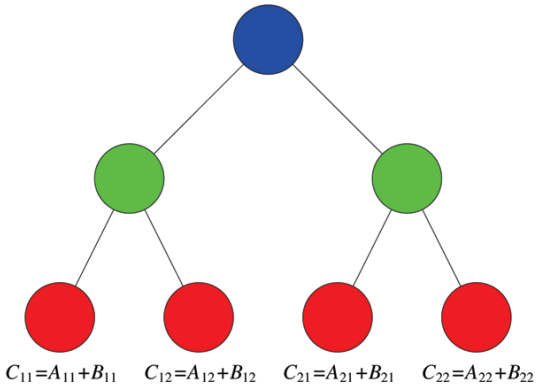

Fig. 1.2 *Recursion tree illustrating the original problem (blue root node), and two levels of recursive blocking resulting in smaller subtasks (two green intermediate nodes and four red leaf nodes).*

the nodes in the recursion tree, and together they define the so-called conquer phase in the recursion. Since the leaf nodes are usually designed to do level 3 operations, this combination of recursion and blocking reduces the overhead cost of recursion to a tiny and acceptable level. However, knowledge of the L1 cache size is required to select an appropriate block size for a given platform. The block size is chosen so that submatrices associated with a leaf computation fit in L1 cache. The recursive blocked algorithms are expressed in divide-and-conquer style, and most of the computations are performed usually in GEMM operations of variable-sized squarish blocks. These operations can execute practically at the peak obtainable rate. The recursive blocking leads to algorithms with fewer tuning parameters than their standard counterparts. This will all be illustrated throughout the paper.

In our simple example, we have chosen the blocking parameter to be a function of n. Ideally, one would like to terminate the recursion when the submatrix operands of addition fit comfortably into the L1 cache. Then the recursive blocked matrix add algorithm becomes more general. It works for any n, and as n increases the number of nodes in the recursion tree increases. In more complicated examples, computations are also performed at the intermediate nodes. Suppose that the L2 cache is two times the size of the L1 cache and that some computation is performed on the green nodes in Figure 1.2 on rectangular matrices of size $n \times n/2$. The operands now fit into L2 cache, and therefore these computations are automatically blocked for the L2 cache. In this example, the recursive blocked algorithm is, and not only potentially, blocked for the L2 cache.

In general, for large n the number of levels in the recursion tree will exceed the number of levels of the memory hierarchy. Due to the recursive blocking each level holds submatrices of smaller and smaller sizes. At some level the submatrices are small enough to fit into a certain level of the memory hierarchy. This is what we mean by saying that the recursive blocked algorithms have the potential to fit every level of a deep memory hierarchy, including registers, cache memories, main memory, and secondary storage.

Recursive blocked algorithms mainly improve on the *temporal locality*, which means that blocks (submatrices) which recently have been accessed will most likely be referenced soon again. For many of our algorithms the use of the recursive block-

ing technique together with new optimized so-called *superscalar kernels* (defined in section 2.1) is enough to reach near to optimal performance. The recursive blocking provides efficient cache and TLB blocking, and the optimized kernels make use of efficient register blocking. For some problems, we can further increase the performance by explicitly improving on the *spatial locality* as well. The goal is now to match the algorithm and the data structure so that blocks (submatrices) near the recently accessed blocks will also be referenced soon. In other words, the storing of matrix blocks in memory should match the data reference pattern of the blocks, and thereby as much as possible minimize data transfers in the memory hierarchy. We use the divide-and-conquer heuristics leading to hybrid data structures that store the blocks recursively. Ultimately, we want to reuse the data as much as possible at each level of the memory hierarchy and thereby minimize the cost.

Similar ideas have been used in related areas by several research groups. As an example, the MIT group headed by Leiserson developed "cache oblivious algorithms" that use recursive layouts and algorithms for automatic data locality for complicated memory hierarchies. Their 1999 FOCS paper addressed sorting, FFT, and matrix transpose [35]. Frigo and Johnson of the MIT group were awarded the third Wilkinson Prize for Numerical Software for their winning work FFTW [34, 33], where these ideas have been most fruitfully applied. FFTW is a library for the efficient computation of the discrete Fourier transform of real and complex data.

Hierarchical data structures like trees have been around for a long time (see Samet [87]). An early example of a recursive data structure in a numerical algorithm is the use of quadtrees and octtrees in N-body simulations. Salmon, Warren, and Winckelmans used these hierarchical data structures when implementing the Barnes–Hut method [7] to achieve data locality in a distributed memory setting [86, 85].

1.4. Preview of Results. This paper gives an overview of recent progress in using recursion as a general technique for producing dense linear algebra software that is efficient on today's memory-tiered computers. In addition to general ideas and techniques, we present detailed case studies of matrix computations. Some of the main points are the following:

- Recursion creates new algorithms for linear algebra software.
- Recursion can be used to express dense linear algebra algorithms entirely in terms of level 3 BLAS like matrix-matrix operations.
- Recursion introduces an automatic variable blocking that targets every level of a deep memory hierarchy.
- Recursive blocking can also be used to define data formats for storing block-partitioned matrices. These formats generalize standard matrix formats for general matrices, and generalize both standard packed and full matrix formats for triangular and symmetric matrices.

In addition, we describe new algorithms and library software for level 3 BLAS, matrix factorizations, and the solution of general linear systems and common matrix equations. The associated software is robust and displays excellent performance on a variety of high-performance platforms. Table 1.1 summarizes the architecture characteristics of the machines considered, including the processor cycle time, theoretical peak performance, sizes of L1 and L2 cache memories, and the number of entries in the TLB. The last column shows the BLAS routines used in our numerical tests. All algorithms and comparisons on a given architecture use the same standard nonrecursive BLAS implementations, except in section 2.3, where we discuss different implementations of GEMM.

Table 1.1 *Architecture characteristics and BLAS library summary.*

Architecture platform	Speed (MHz)	Peak perf. (Mflops/s)	L1 cache (data, kB)	L2 cache (shared, kB)	TLB (entries)	BLAS used
IBM PowerPC 604	112	224	16	512	128	ESSL [57]
IBM PowerPC 604e	332	664	32	256	128	ESSL
IBM Power2	120	480	128	—	512	ESSL
IBM Power2	160	640	128	—	512	ESSL
IBM Power3	200	800	64	4096	256	ESSL
IBM Power3	375	1500	64	4096	256	ESSL
Intel Pentium III	550	550	16	512	32	ATLAS [100]
MIPS R10000	195	390	32	4096	64	SCSL [89]

Many of the sample performance results to be shown later are close to (i.e., within 50% to 90% of) the peak attainable performance of the machines for large enough problems, showing the effectiveness of the techniques described. Park, Hong, and Prasanna [77] have recently provided further support concerning minimizing L1 and L2 cache and TLB misses.

1.5. Organization of This Paper. Section 2 introduces recursive blocked algorithms, starting with basic recursive blocked splittings and templates for some level 3 BLAS and matrix factorizations. Section 2.2 gives a survey of recursive blocked data structures, including rectangular and triangular data formats. In section 2.3, some performance results of the GEMM operation are presented that illustrate the potential of combining recursive blocking and hybrid data structures.

In section 3, a recursive blocked Cholesky factorization using packed recursive blocked data storage is discussed. Section 4 presents a survey of recursive blocked algorithms for the QR factorization (see section 4.1) and for solving general linear systems $AX = B$. In section 4.2, both the solution of over- and underdetermined linear systems are considered. Section 5 is devoted to a survey of recent work on recursive blocked solvers for triangular matrix equations and related condition estimation problems. One-sided and coupled Sylvester-type matrix equations are discussed in section 5.1. Two-sided and generalized Sylvester and Lyapunov matrix equations are treated in section 5.2. In section 5.3, the use of matrix equation solvers in condition estimation (Sep-estimation) is discussed. In sections 4 and 5, recursive blocked algorithms are applied to data stored in standard data layout. Here, only marginal additional gain in speed has been observed when the matrices are stored in a recursive blocked data format [75, 59].

Section 6 discusses the use of blocked standard algorithms with hybrid blocked data formats to manage deep memory hierarchies. The presentation and discussion of related and complementary work continues in section 7. In section 7.1, the topic is related work on recursive algorithms, hybrid data structures, and library software implementations. Section 7.2 briefly presents and discusses complementary work that automatically generates software, as well as some related compiler work. Finally, in section 8, we give some concluding remarks.

2. Recursive Blocked Templates and Hybrid Data Structures. Here, we present the general ideas and techniques that later will be illustrated by case studies in sections 3.3 and 4–7.

$$
\begin{aligned}
&\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} + \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} \\
&= \begin{bmatrix} \begin{bmatrix} C_{11} & C_{12} \end{bmatrix} + \begin{bmatrix} A_{11} & A_{12} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} \\ \\ \begin{bmatrix} C_{21} & C_{22} \end{bmatrix} + \begin{bmatrix} A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} \end{bmatrix} \\
&= \begin{bmatrix} \begin{bmatrix} C_{11} \\ C_{21} \end{bmatrix} + \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} \\ B_{21} \end{bmatrix}, & \begin{bmatrix} C_{12} \\ C_{22} \end{bmatrix} + \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{12} \\ B_{22} \end{bmatrix} \end{bmatrix} \\
&= \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} + \begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \end{bmatrix} + \begin{bmatrix} A_{12} \\ A_{22} \end{bmatrix} \begin{bmatrix} B_{21} & B_{22} \end{bmatrix}
\end{aligned}
$$

Fig. 2.1 *Splitting the matrix multiplication on the (m, n, k) dimensions (top), and the m, n, and k dimensions, respectively. The resulting GEMM suboperations are shown for the m, n, and k splittings.*

2.1. Recursive Blocked Templates for Level 3 BLAS and Matrix Factorizations. We start by introducing basic recursive blocked splittings and templates for some typical matrix operations, namely, the level 3 BLAS operations general matrix multiply and add (GEMM) and triangular solve with multiple right-hand sides (TRSM), and a generic matrix factorization. The splittings generate new smaller subproblems (tasks) in a recursion tree. For each of these subproblems a recursive blocked template is applied, which in turn generates new tasks, etc. Typically, the recursion is terminated when the new problem sizes are smaller than a certain block size, *blksz*, which is chosen such that at least the submatrices involved in the current subproblem fit in the L1 cache memory. For the solution of the small-sized leaf problems of the recursion tree, we apply novel portable high-performance kernels based on reliable (standard) algorithms.

GEMM Operation. Without loss of generality, we consider $C = C + AB$, which is the no-transpose case of $C \leftarrow \alpha \mathrm{op}(A)\mathrm{op}(B) + \beta C$, with $\alpha = \beta = 1$ and, e.g., $\mathrm{op}(A)$ denotes the matrix A or its transpose A^T. We remark that $\alpha = -1$ almost always occurs in matrix factorization updates. Here, C is $m \times n$, A is $m \times k$, and B is $k \times n$.

Since there are three problem dimensions m, n, and k that define a valid GEMM operation, the splitting of A, B, and C can take place in one or more of them. In Figure 2.1, we show four different splittings (out of seven possible). The topmost corresponds to splitting in all three problem dimensions simultaneously, leading to eight smaller GEMM operations. The other three correspond to splitting the matrix multiplication in the m, n, and k dimensions, respectively. Splitting in the m (or n) dimension leads to two independent GEMM operations. Splitting in the k dimension leads to two "rank-$k/2$" updates, which typically could be performed as two successive GEMM operations.

The remaining splittings correspond to splitting in two of the three problem dimensions (see Figure 2.2). Each of the partitionings leads to four smaller GEMM operations.

Depending on which dimension(s) a recursive blocked splitting is performed, two, four, or eight new subproblems are generated. The more tasks we generate in the "divide" part, the smaller are the submatrices involved. By splitting the m, n, and k dimensions simultaneously, we do a *splitting by breadth* and generate eight new tasks.

$$
\begin{aligned}
&\left[\begin{array}{c|c} C_{11} & C_{12} \\ \hline C_{21} & C_{22} \end{array}\right] + \left[\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right] \left[\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array}\right] \\
&= \left[\begin{array}{cc} C_{11} & C_{12} \\ C_{21} & C_{22} \end{array}\right] + \left[\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right] \left[\begin{array}{cc} B_{11} & B_{12} \\ B_{21} & B_{22} \end{array}\right] \\
&= \left[\begin{array}{c|c} C_{11} & C_{12} \\ C_{21} & C_{22} \end{array}\right] + \left[\begin{array}{c|c} A_{11} & A_{12} \\ A_{21} & A_{22} \end{array}\right] \left[\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array}\right]
\end{aligned}
$$

Fig. 2.2 *Splitting the matrix multiplication on the (m,n), (m,k), and (n,k) dimensions, respectively.*

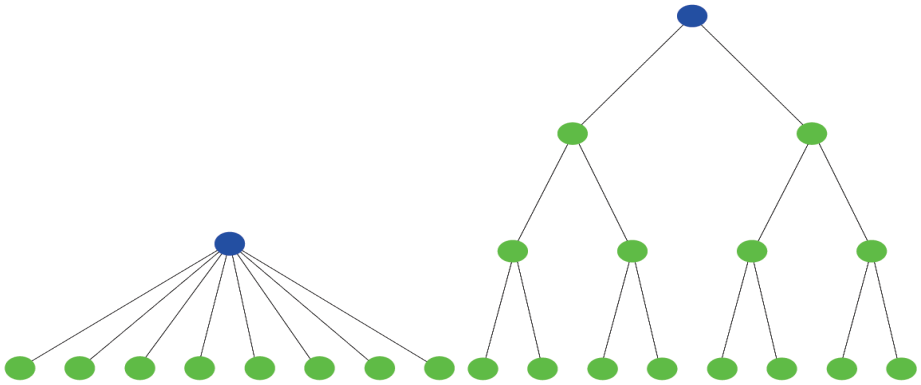

Fig. 2.3 *Recursion trees illustrating splitting by breadth (left) and depth (right).*

On the other hand, by splitting only one of the three dimensions, two new subproblems are generated, and we need to recursively repeat the splitting twice more to get eight tasks of similar size as one step of the splitting by breadth. Accordingly, we call this *splitting by depth*. In Figure 2.3, the recursion trees associated with splitting a task into eight subtasks by breadth and depth, respectively, are displayed. Splitting in two of the three dimensions can be seen as a hybrid of the two, since four new tasks in each splitting are generated. Typically, one makes the choice of splitting to generate "squarish" subproblems, i.e., the ratio between the number of operations made on subblocks and the number of subblocks is maintained as high as possible. Nevertheless, the "conquer" part in GEMM is trivial, since the submatrix additions of the results are made implicitly ("on the fly") by the leaf operations. The subblocks fitting in L1 cache guarantee that there is no or only small performance differences between different transpose arguments of the GEMM operation (AB, A^TB, AB^T, A^TB^T). For a thorough explanation of these kernels, see [47] and [51].

Notice that if $m = n = k > 1$, or two of them are equal, there is a choice on which dimension to split. Some experimental results have shown that this choice is not significant, since different orderings affect only a few levels of the recursion tree, and so further down the recursion tree, the impact of previous choices vanishes. In our implementations, we have focused on square subblocks and mostly square matrices.

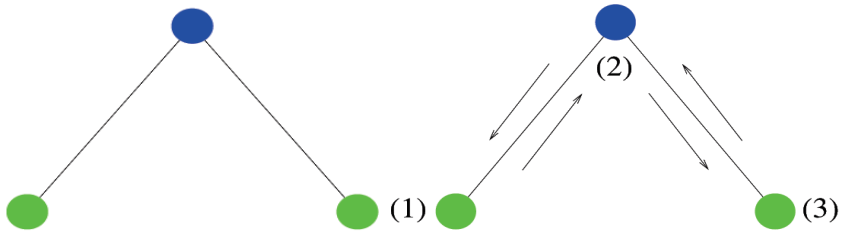

Fig. 2.4 *Splittings defining independent tasks (left) and dependent tasks (right). The right-hand splitting defines a critical path of subtasks:* (1), (2), (3).

TRSM Operation. First, we consider solving $AX = C$, where X overwrites C. A of size $m \times m$ is upper triangular, and C and X are $m \times n$. Depending on m and n, there are several alternatives for doing a recursive splitting. Two of them are illustrated below.

Case 1 ($1 \leq m \leq n/2$). Split C by columns only,

$$A \begin{bmatrix} X_1 & X_2 \end{bmatrix} = \begin{bmatrix} C_1 & C_2 \end{bmatrix},$$

or, equivalently,

$$\begin{aligned} AX_1 &= C_1, \\ AX_2 &= C_2. \end{aligned}$$

Case 2 ($1 \leq n \leq m/2$). Split A, which is assumed to be upper triangular, by rows and columns. Since the number of right-hand sides n is much smaller than m, C is split by rows only,

$$\begin{bmatrix} A_{11} & A_{12} \\ & A_{22} \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \end{bmatrix} = \begin{bmatrix} C_1 \\ C_2 \end{bmatrix},$$

or, equivalently,

$$\begin{aligned} A_{11}X_1 &= C_1 - A_{12}X_2, \\ A_{22}X_2 &= C_2. \end{aligned}$$

The two splittings above are fundamental in all types of triangular solve (or multiply) operations and illustrate that a *problem is split into two subproblems with dependent and independent tasks*, respectively. In Case 1, a splitting is applied only to C, the right-hand sides, and we obtain two similar TRSM operations that can be solved independently and concurrently (illustrated in Figure 2.4 (left)). In Case 2, we first have to (1) solve for X_2 and (2) update the right-hand side C_1 with respect to X_2, which is a GEMM operation, before (3) solving for X_1. The splitting of A imposes a *critical path* at the block level that any algorithm (recursive or nonrecursive) has to respect (illustrated in Figure 2.4 (right)).

There is also a Case 3 ($n/2 < m < 2n$), when all matrices involved are split by rows and columns leading to four subproblems. We refer to sections 5.1 and 5.2 for a more complete illustration of different recursive splittings applied to solving triangular Sylvester-type matrix equations. Moreover, the other variants of the TRSM operation ($\mathrm{op}(A)X = C$ or $X\mathrm{op}(A) = C$, where $\mathrm{op}(A)$ is A or A^T, with A upper or lower triangular) are treated similarly.

Matrix Factorization. We introduce a recursion template for a generic one-sided matrix factorization (typified by Cholesky, LU, and QR) as follows.

Recursively factor A:

1. *Partition*

$$A \equiv \left[\begin{array}{cc} A_1 & A_2 \end{array} \right], \quad \text{where } A_1 \equiv \left[\begin{array}{c} A_{11} \\ A_{21} \end{array} \right] \text{ and } A_2 \equiv \left[\begin{array}{c} A_{12} \\ A_{22} \end{array} \right].$$

2. *Recursively factor* A_1 (or A_{11}).
3. *Apply* resulting transformations to $A_2 \equiv \left[\begin{smallmatrix} A_{12} \\ A_{22} \end{smallmatrix} \right]$.
4. *Recursively factor* A_{22}.

In the algorithm, the template is recursively applied to the two smaller subproblems (factorizations in steps 2 and 4). As for the triangular solve operation, the recursion template results in a splitting defining a critical path of dependent subtasks (see Figure 2.4 (right)).

We remark that in cases when only one of the two matrix dimensions is split, as for the QR factorization, the recursion is stopped when the overhead from recursion exceeds the gain from using a nonrecursive algorithm (see section 4.1). When both dimensions are split, as for the Cholesky factorization, the recursive blocking is continued until at least the submatrices involved in the current subfactorization fit in the L1 cache memory.

We note that the apply or update operation (step 3) can be performed either by calling recursive blocked or standard blocked algorithms/implementations of level 3 BLAS (or similar operations). In the recursive blocked algorithms presented below, one can make use of both alternatives. By symmetry, the apply operation of step 3 works in theory also for the Cholesky factorization. In practice, however, the transformations from the factorization in step 2 is applied to $[A_{21}, A_{22}]$ (see section 3).

Superscalar Kernels. Each level 3 BLAS algorithm does more or less the following: It takes the input matrix operands and performs a partitioning of these operands into submatrices. These are usually called blocks and fit comfortably into the L1 cache. Usually these blocks are copied into contiguous storage buffers. An associated implementation operates on these blocks and performs the BLAS operation on submatrices of the partitioning. This associated program is called the kernel routine of the level 3 BLAS.

So kernel routines are very highly performing programs working on matrix operands optimally prepared for excellent performance in L1 cache. These kernel routines were the building blocks of ESSL BLAS. However, ESSL went further and constructed routines for various factorization algorithms, e.g., $LU = PA$, $LL^T = A$, $QR = A$, etc. These then are superscalar kernels. Our work has produced designs for new and many more superscalar kernels that are applied to the leaf nodes in the recursion tree. Indeed, some of the superscalar kernels make use of recursion as well [62, 63]. All superscalar kernels are written in Fortran using register and cache blocking techniques such as loop unrolling. For each recursive blocked algorithm the same superscalar kernels are used on all platforms. Currently, they are optimized with a generic superscalar architecture in mind and show very good performance on several different platforms. We remark that it would be possible to optimize these superscalar kernels with respect to different architecture features and possibly gain some additional performance. The generic superscalar kernels make the recursive blocked algorithms portable across different computer systems.

2.2. Recursive Blocked Data Structures. In [46], we introduced a new set of data formats for storing block-partitioned dense matrices. The set is a hybrid of two addressing techniques. At the block level each submatrix is stored in the standard column-major (or row-major) order and the size of each block is constrained so that one or a few of them will simultaneously fit in L1 cache. Blocks stored in this fashion are typically operands for the kernel routines. To allow for efficient utilization of a memory hierarchy, including efficient cache reuse, the blocks themselves can be stored recursively. Due to the regularity of dense linear algebra computations, it is in principle enough to consider only two recursive matrix formats, namely, the *rectangle* and the *isosceles triangle*. Some related work on recursive data layouts is mentioned in section 7.

2.2.1. Rectangular Recursive Data Format. The metrics of a matrix A of size $m \times n$ stored in ordinary column-major order (Fortran format) include m, n, and the leading dimension lda. The latter is used for a proper specification of a subarray of A. For block-partitioning of A we also use two parameters specifying the block sizes, mb and nb, where $1 \leq mb \leq m$ and $1 \leq nb \leq m$. A block-partitioned A consists of $p \cdot q$ blocks A_{ij} of size $mb \times nb$, where $p = \lceil m/mb \rceil$ and $q = \lceil n/nb \rceil$. We use the convention that the last block row and/or block column are padded with zero elements when m and/or n are not multiples of mb and nb. However, little or no computations on these zero elements are performed. Each submatrix A_{ij} is stored in column-major or row-major order. Notice that padding leads to many economies in code production, e.g., no fix-up code is required in the kernel routines.

Now, these submatrices can be ordered in $(p \cdot q)!$ different ways. A recursive ordering (use of the Hilbert heuristic of a one-dimensional tour through a two-dimensional object [84]) has the potential of matching the memory hierarchies of today's high performance computing systems and is therefore an effective ordering when performing linear algebra operations on the matrix. A recursive blocked format allows for the possibility of maintaining the two-dimensional data locality at every level of the one-dimensional tiered memory structure, while the block row and block column orderings only maintain data locality at a submatrix level. Irrespective of how the block sizes are set, the block row or block column format can only automatically match one level of the memory hierarchy, e.g., L1 cache.

The recursive block ordering is determined by always dividing the largest dimension of the rectangular submatrix. The choice when there is a tie leads to two formats: *recursive block row* (RBR), always divide the row dimension; and *recursive block column* (RBC), always divide the column dimension. When dividing an odd number of rows the middle row is assigned to the block at the bottom. For an odd number of columns, the middle one is assigned to the block to the right. The reason is that the block to the right or at the bottom may contain submatrices that are not entirely filled, so this strategy keeps the difference in number of used elements between the two blocks after the splitting to a minimum. Our RBR blocking is a variant of the Z-Morton ordering [102, 15, 84]. See Figure 2.5 for examples of Z-Morton ordering. Indeed, they are the same when the block matrix dimensions p and q both are a power of 2, which means that all blocks are equal-sized at a fixed level of the recursive blocking.

For illustration we consider A of size 800×480 and $mb = 100$, $nb = 60$, giving $p = q = 8$; i.e., all blocks are filled with elements from A. Using a block column format, the blocks are mapped as in the top matrix labeled (BC) of Figure 2.5. The numbers 0–63 denote contiguous blocks in memory. Using the RBR format for assigning contiguous

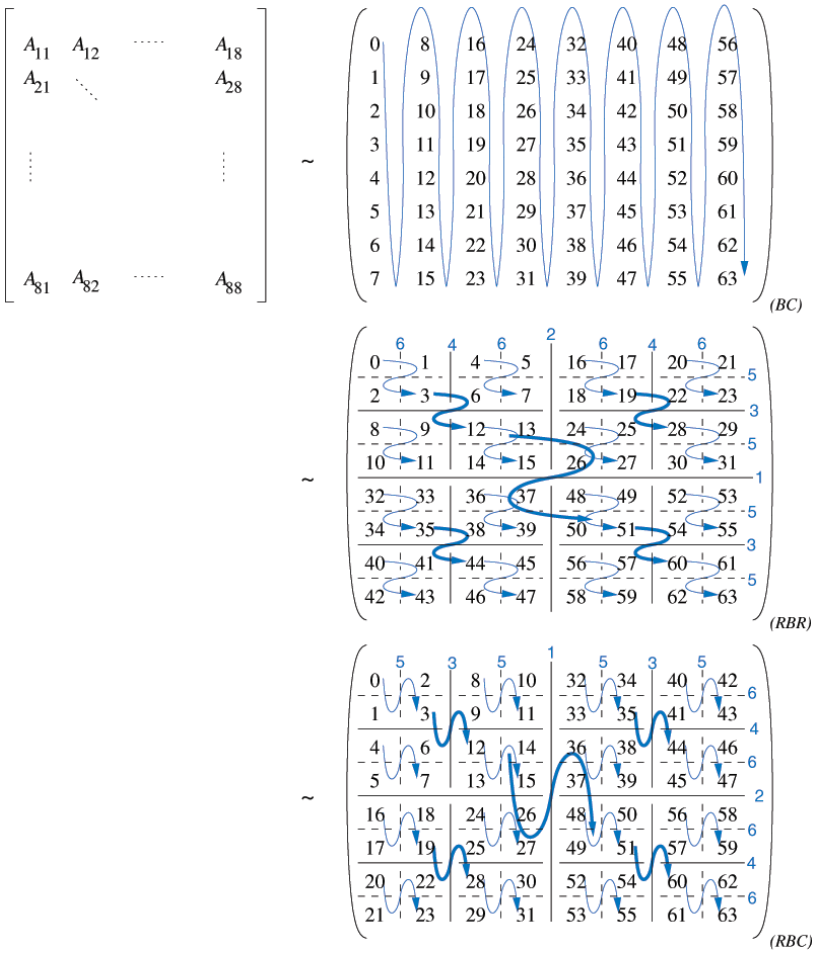

Fig. 2.5 *The mapping of an* 8×8 *block matrix in block column (BC) order, recursive block row (RBR) order, and recursive block column (RBC) order. The three levels of recursive splitting are labeled* 1-2, 3-4, *and* 5-6.

blocks in memory, the first splitting occurs after block row four and the block numbers 0–31 are assigned to the upper part and 32–63 to the lower part, respectively. Since the number of block columns (4) is greater than the number of block rows (2) in the upper part, the next splitting is vertical, assigning block numbers 0–15 to the left-hand part and 16–31 to the right-hand part. These submatrices are now square so their row dimension is split. The complete block assignment associated with the RBR format is displayed in the matrix labeled (RBR) of Figure 2.5. The procedure is similar when applying the RBC assignment of contiguous blocks, except when splitting square submatrices, which is now done by splitting the column dimension (see the matrix labeled (RBC) of Figure 2.5). In the figure, we have also marked in which order the recursive splittings are done for the recursive RBR and RBC formats. For this example, the RBR and RBC orderings correspond to the Z-Morton and the reflected-N-Morton space filling orderings, respectively. Again, see Figure 2.5, references [102, 15, 84], and section 7.1 for more details. For arbitrary m, n, mb, and nb, our recursive blocked orderings are combinations of these two Morton orderings.

The choice of mb and nb is crucial and closely linked to the memory hierarchy of the target architecture. In the extreme case of $mb = m, nb = n$, the entire matrix is stored in one block, which corresponds to the conventional column-major and row-major orderings with their deficiencies like bad data locality in either the row or the column dimension. When $mb = nb = 1$, this means that recursive splittings are applied down to single elements. This results in good data locality, but the submatrix operations become very inefficient since the kernels will only operate on single elements. For example, register blocking is prohibited. The best choice of mb and nb depends on the size of the L1 cache. One should strive for fitting one or a few submatrices in L1 cache. The consequence of having bad data locality in one dimension of the submatrix does not hinder performance, since the L1 cache is approximately random access for data stored contiguously. Thus, we have linear addressing in the kernels, which simplifies loop unrolling, preloading, prefetching, and register blocking. The kernels can also feature level 3 prefetching; i.e., they can make use of the *volume-to-surface effect* between the number of floating-point operations and the number of memory accesses for level 3 BLAS operations. In principle, the program loads the next set of operands into L1 cache while the prior set is being used in the computations. The cost to calculate the starting address of an element block is $O(\log(p \cdot q))$ (binary search in two block dimensions). However, this information may be stored in tables, which gives greater flexibility in the placement of the blocks and constant time addressing.

Another benefit of the tables is that the space allocated for each block, S, may be greater than the space required; i.e., the block addresses may be padded so that cache coherency problems may be avoided. For example, the allocation strategy may be to let all blocks begin at a line or page boundary.

2.2.2. Triangular Recursive Data Format. Since all one-sided matrix factorizations can be expressed in terms of rectangular and isosceles triangular matrices, there are two cases to consider. We just consider the isosceles case. For an isosceles right triangle of order n, the splitting procedure resembles the rectangular case. Let $nb \times nb$ be the size of the submatrices, giving a block-square triangular A consisting of $q(q+1)/2$ blocks where $q = \lceil n/nb \rceil$. Now divide the triangle into one subrectangle and two isosceles subtriangles. For a lower triangle, the upper left triangle is assigned block numbers 0 to $\lfloor q/2 \rfloor \lfloor q/2+1 \rfloor /2-1$, the lower right triangle is assigned block numbers $\lfloor q/2 \rfloor \lfloor q/2+1 \rfloor /2 + \lfloor q/2 \rfloor \lceil q/2 \rceil$ to $q(q+1)/2-1$, and the blocks inside the rectangle will use block numbers $\lfloor q/2 \rfloor \lfloor q/2+1 \rfloor /2$ to $\lfloor q/2 \rfloor \lfloor q/2+1 \rfloor /2 + \lfloor q/2 \rfloor \lceil q/2 \rceil -1$. The interior ordering of the blocks in the triangles are determined by applying the algorithm recursively, and the block ordering is either RBR or RBC as in the rectangular case.

Figure 2.6 illustrates the triangular recursive blocked orderings for triangular matrices of order 320, and block size $nb = 80$, giving $q = 4$. The diagonal blocks 0, 2, 7, and 9 are stored in the conventional full format. We use full format since it is easier to write high-performance kernel routines using this format. The blocks in the last block row or block column are possibly padded. The zero blocks in the upper or lower parts, respectively, are not stored, so the space overhead is linear to the matrix order.

From Figure 2.6 one can see that the RBR ordering of a symmetric matrix in lower triangular storage format is equivalent to the RBC ordering of a symmetric matrix in upper triangular storage format, and vice versa.

2.3. Some GEMM Performance Results. We end the survey of recursive blocked templates and hybrid data structures by the first case study. Some performance

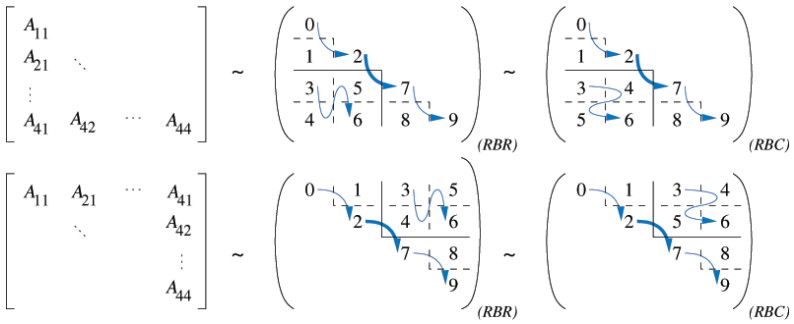

Fig. 2.6 *The mapping of a 4 × 4 block triangular matrix in block column (BC) order, recursive block row (RBR) order, and recursive block column (RBC) order. Two levels of recursive splitting are used.*

results of the GEMM operation that illustrate the potential of this particular recursive approach [46, 47] are presented and discussed.

Figure 2.7 shows the performance of an explicitly tuned two-level blocked algorithm and a recursive GEMM algorithm executing on an IBM PowerPC 604. Even though this architecture is fairly old it is interesting since it illustrates a system with a rather complex memory hierarchy, where the use of floating-point units is much faster than accessing the cache memories. Both routines use the same superscalar kernel that prefetches, register preloads, does 4×4 unrolling, and works on 16×16 submatrices which utilize standard row/column-major addressing on contiguous stored blocks [46, 47]. The performance results, which are quite good, do not account for any data copying; i.e., the blocks are initially stored in column block order and recursive block order, respectively. The recursive algorithm performs around 7% better than the multilevel blocked algorithm for large enough problems (500×500), even though the tuning is much less explicit (only L1 cache versus L1 and L2 cache tuning for the multiblocked algorithm).

Figure 2.7 also displays ATLAS performance results for the same operations. ATLAS is a project on automated empirical optimization of software for linear algebra [100] and is described further in section 7.2. The results show that the recursive approach is asymptotically at least as good as explicit multilevel blocking and much better than the ATLAS experimental approach for the PowerPC 604. This extreme result has not been observed on other platforms but was a motivation for further studies in using recursion for dense linear algebra computations. At the time this study was made, ATLAS used explicit multilevel-type blocking techniques and had difficulties dealing with this architecture (see Table 1.1). For current algorithms used in ATLAS we refer to the discussion in section 7.2. We remark that without architecture-based optimizations matrix computations seldom reach 50% of the theoretical peak performance on PowerPC 604 platforms, mainly due to the performance imbalance between the memory system and the CPU.

The recursion task tree facilitates parallelization on shared memory machines. One can simply divide the tree into subtrees and let different processes or threads (lightweight processes) execute on different subtrees. Here, we use a single thread for each processor and divide the tree into one subtree more than the number of threads. The leftover subtree is divided among the threads when they become ready for additional work. This way of scheduling enables good load balancing on nondedicated machines.

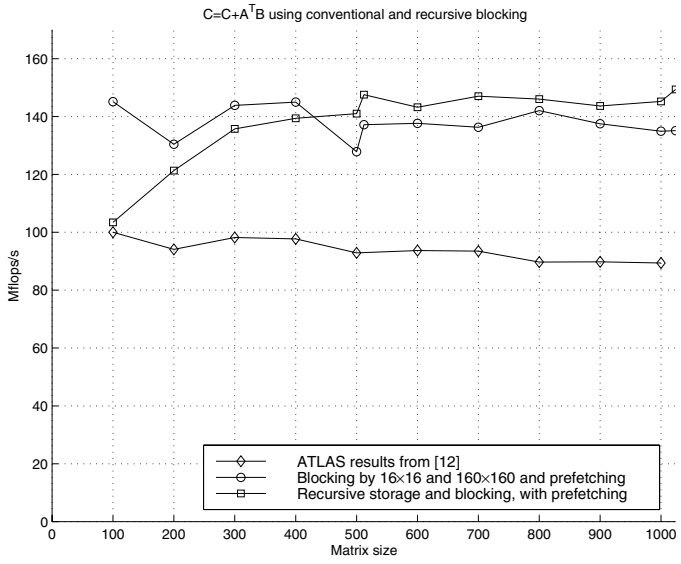

Fig. 2.7 *Performance of two hierarchical blocking strategies: Explicit multilevel and recursive blocking (IBM PowerPC* 604 *platform).*

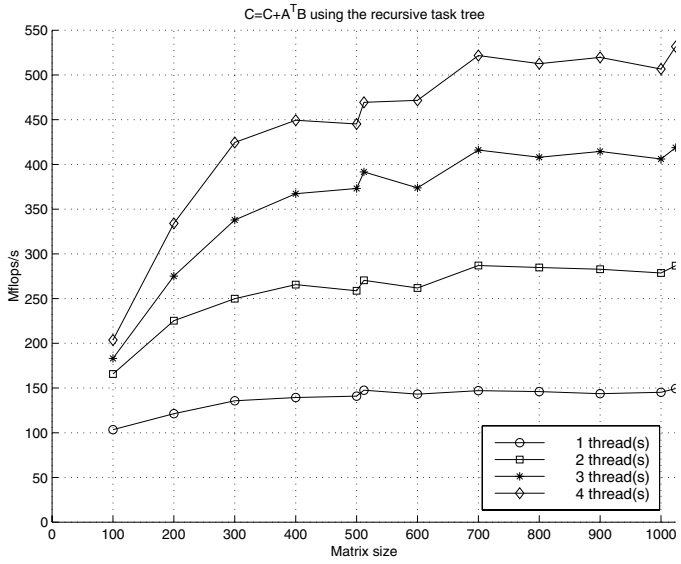

Fig. 2.8 *Scheduling the recursive blocked matrix multiply algorithm on several threads (IBM PowerPC* 604 *platform).*

Performance results for multiple threads and the recursive task tree approach are shown in Figure 2.8 [51]. This algorithm scales well. One reason is that the recursion task tree automatically keeps data references local to each thread.

3. Recursive Blocked Cholesky Factorization for Matrices in Packed Format. The second case study concerns new efficient Cholesky factorization algorithms. In [3], a new recursive packed format for Cholesky factorization was described. It is a variant of the triangular format described in [46] and suggested in [43]. The advantage of using packed formats instead of a full format is the memory savings possible when working with large matrices. The disadvantage is that the use of high-performance standard library routines, such as GEMM, is inhibited. This is because level 3 implementations for packed data formats are not present in LAPACK and some other libraries. One can combine blocking with recursion to produce a practical implementation of a new algorithm for blocked packed Cholesky. The algorithm first transforms standard packed lower format to packed recursive lower row format. Then, during execution, the algorithm calls only standard GEMM and level 3 kernel routines. This method has the benefit of being transparent to the user. No extra assumptions about the input matrix needs to be made. This is important since an algorithm like Cholesky using a new data format would not work on previous software implementations.

Rationale. Existing codes for the Cholesky factorization and the factorization $A = LDL^T$ use either full storage or packed storage data format. ESSL [56] and LAPACK [4] as well as many other libraries support both data formats so a user can choose either for his application. For performance reasons, users today generally use the full data format to represent the symmetric matrices. Nonetheless, saving half the storage is an important consideration especially for those applications which will run with packed storage and fail to run with full storage.

The idea behind recursive factorization of a symmetric matrix stored in packed recursive format is simple: Given AP holding symmetric A in lower packed storage mode, overwrite AP with A in the recursive packed row format. Next, execute the new recursive level 3 Cholesky algorithm. Even when one includes the cost of converting the data from conventional packed to recursive packed format, the performance turns out to be better than LAPACK's level 3 routine DPOTRF for full storage format.

3.1. Packed Recursive Blocked Data Storage and Algorithm. The new recursive packed format was first presented in [3] and is based on the formats described in section 2.2.2. An algorithm which transforms from conventional packed format to recursive packed format can be found in [48] and [3]. The packed recursive data format is a hybrid triangular format consisting of $n-1$ full format rectangles of varying sizes and n triangles of size 1×1 on the diagonal. The format uses the same amount of data storage as the ordinary packed triangular format, i.e., $n(n+1)/2$. Since the rectangles (square submatrices) are in full format it is possible to use high-performance level 3 BLAS on these square submatrices. The difference between the packed and the packed recursive formats is shown in Figure 3.1 for a matrix of order 7.

Notice that the triangles are split into two triangles of sizes $n_1 = n/2$ and $n_2 = n - n_1$ and a rectangle of size $n_2 \times n_1$ for lower format and $n_1 \times n_2$ for upper format. The elements in the upper left triangle are stored first, the elements in the rectangle follows, and the elements in the lower right triangle are stored last. The order of the elements in each triangle is again determined by the recursive scheme of dividing the sides n_1 and n_2 by two and ordering these sets of points in the order triangle, rectangle, triangle. The elements in the rectangle are stored in full format, either by row or by column.

1	2	4	7	11	16	22
	3	5	8	12	17	23
		6	9	13	18	24
			10	14	19	25
				15	20	26
					21	27
						28

Packed upper

1	2	3	7	10	13	16
	4	5	8	11	14	17
		6	9	12	15	18
			19	20	22	24
				21	23	25
					26	27
						28

Packed recursive upper

Fig. 3.1 *Memory indices for 7×7 upper triangular matrix stored in traditional packed format and recursive packed format.*

The recursive formulation of the algorithm is straightforwardly derived from the block factorization of a positive definite matrix A,

$$A \equiv \begin{bmatrix} A_{11} & A_{21}^T \\ A_{21} & A_{22} \end{bmatrix} = LL^T \equiv \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} L_{11}^T & L_{21}^T \\ 0 & L_{22} \end{bmatrix},$$

which consists of two Cholesky factorizations (3.1), (3.4), one triangular system solve with multiple right-hand sides (3.2), and one symmetric rank-k update (3.3),

$$A_{11} = L_{11}L_{11}^T, \tag{3.1}$$

$$L_{21}L_{11}^T = A_{21}, \tag{3.2}$$

$$\tilde{A}_{22} = A_{22} - L_{21}L_{21}^T, \tag{3.3}$$

$$\tilde{A}_{22} = L_{22}L_{22}^T. \tag{3.4}$$

These equations build the *recursive template* for the recursive Cholesky factorization. After recursively solving for L_{11}, a recursive implementation of TRSM is used to solve for L_{21}. Then a recursive symmetric rank-k update (SYRK) of A_{22} occurs before L_{22} is recursively Cholesky factored. The need of the recursive TRSM and SYRK stems from the recursive packed format. The factorization algorithm calls TRSM and SYRK with triangular matrix operands stored in recursive packed format, and with rectangular matrix operands stored in full format. Dividing the recursive packed matrices in TRSM and SYRK gives rise to two recursive packed triangular matrices and a rectangular matrix stored in full format, which becomes an argument to GEMM.

In the implementation presented in [3], recursion proceeds down to the element level. Because of the overhead of the recursive calls, the implementation has a significant performance loss for small problems and a lesser loss for larger problems. For large problems most of the computation is performed in high-performance GEMM operations.

The implementation described in [48], however, combines blocking and recursion to produce a blocked version of the recursive algorithm by only applying recursion down to a fixed block size less than *blksz*. To solve leaf problems of size at least one less than *blksz*, algorithmic techniques, such as register and L1 cache blocking [46], are used to produce optimized unrolled superscalar kernel routines. These techniques are used both by the Cholesky factorization routine and the recursive TRSM and SYRK routines.

By providing these superscalar kernel routines, the procedure call overhead for small problems is significantly decreased. Also, one can overcome the overhead of the

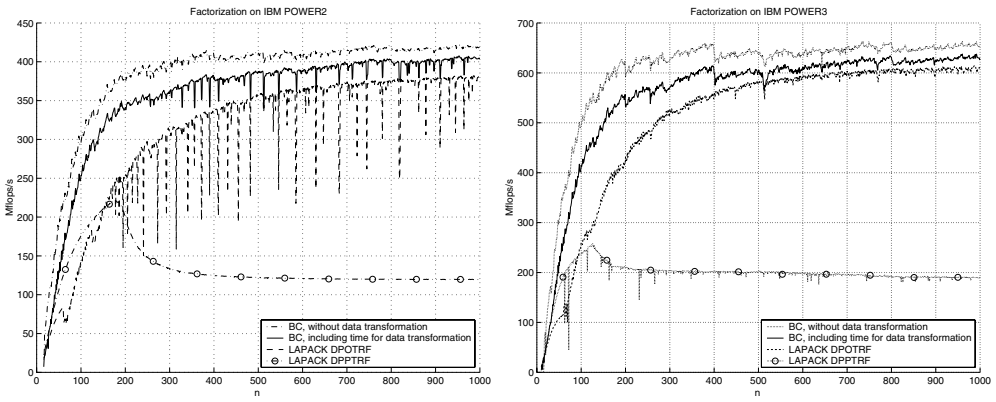

Fig. 3.2 *The performance for Cholesky factorization on the* 120 *MHz IBM Power2 (left) and* 200 *MHz Power3 (right).*

nonlinear addressing of the recursive data format. The operands of the kernel routines for factorization, triangular solve, and symmetric rank-k update are stored in recursive packed format. The two latter use mixed format, where some operands are in full format. In this implementation, two types of kernel routines have been considered. The difference is how they deal with the nonlinear addressing of the packed recursive triangles. The first one is called a mapping kernel, and this technique is used for the Cholesky factorization kernel. Recall that this kernel operates solely on a triangular matrix, which is stored in packed recursive lower format. The ratio of the number of triangular matrix accesses to the number of operations is small, so the performance loss of copying the triangle to a full matrix in order to simplify addressing is not feasible. Instead, an address map of the elements' structure in memory is constructed in advance. Now, for every problem size n, there will only be two kernel problem sizes, so only two maps need to be generated, one for problem sizes $n_i + 1$ and one for problem sizes n_i, where $n_i < blksz$. These maps are initialized before the recursive Cholesky algorithm starts.

For the TRSM and SYRK kernels, more floating-point operations are performed, which reduce the ratio of the number of accesses to the number of operations. This suggests that it could be beneficial to copy the triangle to a buffer, and thereby store the triangle in full array format. In fact, since the number of operations depends on the size of the rectangular operand as well, one can use this size as a threshold. Therefore, if the rectangle is large enough, the triangle is copied to a buffer.

Performance. The use of recursive blocking ensures that the good performance can be maintained also for small problems. Results for both large and small problems for various machines are found in [48] and [3]. The performance of the recursive algorithm, called BC, on a typical RISC processor (Power3) is about 5% better than *full storage* LAPACK routine DPOTRF for large problems, and much better than DPOTRF for small problems, because of the superscalar kernels described partly here and in [48]. See Figure 3.2 for details. The recursive algorithm is much faster (2 to 3.5 times) than the LAPACK routine DPPTRF, which operates on packed storage and hence cannot benefit from a level 3 algorithm. In these tests, the ESSL BLAS (nonrecursive implementations) are used in all four routines. For each algorithm, the best block size has been chosen. For DPOTRF this is $nb = 64$. For BC, $blksz = 32$

is used as the recursion stopping criteria, implying that the blocksize nb for the leaf problems satisfies $16 \leq nb < 32$.

For smaller matrices (below $n = 100$), the observed speedup is a combination of the kernels and the ability to use level 3 kernel routines. Therefore, this is an example of recursive blocking. For larger matrices, the cause of the good performance is due to better memory access patterns and the level 3 BLAS, which are unavailable to the standard packed routine. This is also shown in [3]. The reason that DPOTRF does poorly for say $n \leq 200$ is that the factorization part of DPOTRF is done by the level 2 algorithm DPOTF2.

The recursive algorithm for Cholesky factorization has three attractive features. First, it uses minimal storage. Second, it attains level 3 performance due to mostly calling matrix-matrix multiplication routines during execution. The new algorithm, curve number 3, outperforms the standard LAPACK routines DPPTRF and DPOTRF. Finally, the new code is portable so existing codes can use the new algorithm. A variant of the algorithm presented here is part of the IBM ESSL [56].

4. Recursive Blocked QR Factorization and Linear Systems. The third case study concerns new efficient algorithms based on recursion for computing the QR factorization of a full rank matrix and solving over- and underdetermined systems of equations. In this section, recursive blocking is combined with matrices stored in standard data format.

4.1. QR Factorization. The recursive algorithms for the QR factorization described in [27, 28] have led to highly efficient library software [30], available in ESSL [56]. The algorithm computes the factorization $A = QR$, using Householder transformations $I - \tau u u^T$, where u is a Householder vector and τ is a scalar. The matrix A is $m \times n$, Q is orthogonal $m \times m$, and R is $m \times n$ upper triangular. In practice, Q is not explicitly formed. If requested, Q can be formed from Householder transformations using the storage-efficient compact WY representation $Q = I - YTY^T$ [10, 88]. Here, Y is $m \times n$ upper trapezoidal, containing n Householder vectors, and T is $n \times n$ upper triangular.

If the number of Householder transformations is large, matrix operations involving Q are normally performed as series of operations with $Q_1, Q_2, \ldots, Q_k$, where $Q = Q_1 Q_2 \cdots Q_k$ and $Q_i = I - Y_i T_i Y_i^T$. The reason is that the overhead cost associated with forming the matrix T is much larger than the cost of computing all the T_i's. This issue is further discussed below.

Recursive Algorithm. The basic algorithm, obtained by applying the recursive matrix factorization template (see section 2.1), performs a matrix splitting on the column dimension followed by three matrix computations. That is, A is partitioned as

$$A = \left[\begin{array}{c|c} A_{11} & A_{12} \\ A_{21} & A_{22} \end{array}\right], \tag{4.1}$$

and the left part is factorized by a recursive call

$$Q_1 \begin{bmatrix} R_{11} \\ 0 \end{bmatrix} = \begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix}. \tag{4.2}$$

The right part is updated with respect to the first factorization,

$$\begin{bmatrix} R_{12} \\ \tilde{A}_{22} \end{bmatrix} \longleftarrow Q_1^T \begin{bmatrix} A_{12} \\ A_{22} \end{bmatrix}, \tag{4.3}$$

and, finally, $\tilde{A}_{22}$ is recursively factorized, $\tilde{Q}_2 R_{22} = \tilde{A}_{22}$. The result is

$$R = \begin{bmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{bmatrix} \quad \text{and} \quad Q = Q_1 Q_2, \quad \text{where} \quad Q_2 = \begin{bmatrix} I & 0 \\ 0 & \tilde{Q}_2 \end{bmatrix}. \tag{4.4}$$

In the basic algorithm, the recursion stops when the matrix to be factorized is a single column, and a Householder transformation [39] is applied to that column. In the implementation presented in [30], the recursion is stopped at $n \leq blksz = 4$, and then these columns are factorized using a kernel routine. The kernel factorization is performed as a direct computation, thereby significantly reducing the software overhead otherwise required by using pure recursion.

We remark that in order to efficiently perform the update (the multiplication with Q_1^T), we need to form the compact WY representation $Q_1 = I - Y_1 T_1 Y_1^T$. This makes it possible to express the update phase in the above algorithm in terms of level 3 operations without explicitly forming Q. In the above algorithm, this requires a recursive expression for forming the matrix Q in the compact WY format, i.e., to make it possible to combine one pair of compact WY expressions in each step of the recursion. It turns out that this process itself leads to a level 3 algorithm, with potential for better utilization of the memory hierarchy than the traditional level 2 algorithm for forming the compact WY representation.

Given $Q_1 = I - Y_1 T_1 Y_1^T$ and $Q_2 = I - Y_2 T_2 Y_2^T$, we need to compute Y and T in order to represent $Q = I - YTY^T$. No flops are required to form $Y = \begin{bmatrix} Y_1 & Y_2 \end{bmatrix}$. By substituting $Q_1 = I - Y_1 T_1 Y_1^T$ and $Q_2 = I - Y_2 T_2 Y_2^T$ into (4.4), we have

$$T = \begin{bmatrix} T_1 & -T_1 (Y_1^T Y_2) T_2 \\ 0 & T_2 \end{bmatrix}. \tag{4.5}$$

Matrix Splitting Decisions. Compared to a level 2 algorithm, a block algorithm based on the compact WY representation for Q performs additional flops. The number of extra flops grows cubically with the number of Householder transformations being aggregated, so therefore the computation of large T-matrices should be avoided [28]. In the recursive algorithm, this is accomplished by an appropriate matrix splitting. When n is large, the left part of A is chosen to have nb columns, where nb acts as a blocking parameter. For $n < nb$, the left part of A is chosen to have $\lfloor n/2 \rfloor$ columns. Then T is only computed corresponding to the maximum nb columns wide left block of A and blocks inside the left block, since these are the only T-matrices that are needed in the updates. By doing so, T of size larger than $nb \times nb$ is never formed.

The effect of the blocking parameter is that the algorithm traverses the n-dimension with a fixed block size nb. Within each such block, the block size is recursively halved. Compared to the corresponding LAPACK algorithm, our blocking parameter imposes the same blocking as the level 3 algorithm DGEQRF, but when DGEQRF calls a level 2 routine for factorizing a block column, the recursive algorithm continues to perform level 3 operations on smaller and smaller blocks (tall narrow matrices). It follows that the features of our algorithm could be obtained in DGEQRF by replacing two calls to level 2 algorithms by a single call to the recursive algorithm. However, by using a blocking parameter in the recursive algorithm, there is no need for an extra routine to perform that outer blocking. See page 946 of [29] for details.

Performance. Figure 4.1 presents performance results of the recursive algorithm RGEQRF in relation to the LAPACK routine DGEQRF on a 200 MHz IBM Power3 processor, originally presented in [30]. The graph on the left-hand side shows the

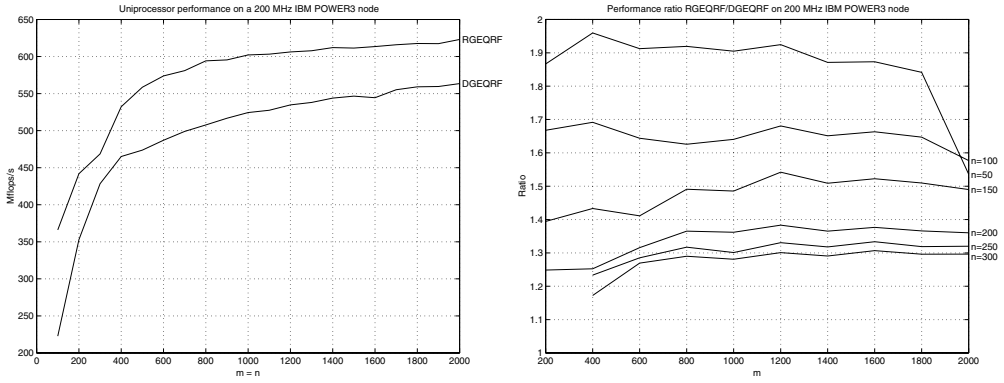

Fig. 4.1 *Performance results in Mflops/s for square matrices (left) and performance ratio for tall, thin matrices (right) for the recursive algorithm RGEQRF and DGEQRF of LAPACK on the* 200 *MHz IBM Power3.*

performance in Mflops/s for both routines. RGEQRF clearly outperforms DGEQRF for all cases tested, and its performance exceeds 600 Mflops/s for square matrices of size at least 1000. The right-hand side shows the performance ratio of RGEQRF to DGEQRF for tall, thin matrices with $m = 200, \ldots, 2000$ and $n = 50, \ldots, 300$. For all cases tested, it outperforms DGEQRF by between 10% and a factor of 2.

Please note that this performance was obtained using a combination of a recursive algorithm and an optimized kernel routine for factorizing matrices of size $m \times 4$. However, [28] presents results for the same algorithm using recursion down to a single column (instead of using an optimized kernel). The use of recursion down to a single column increases the overhead, but the results are still of interest in order to illustrate the effect of the recursion alone. Also this algorithm outperforms LAPACK by around 20% for large matrices and is faster than LAPACK for all square matrices larger than 300 on the 120 MHz IBM Power2 processor. For tall, thin matrices, e.g., with $n \leq 150$, the recursive algorithm is faster for all $m > 200$. Clearly, the recursion leads to an algorithm that outperforms LAPACK for most problems even though an optimized kernel may be needed to compensate for the extra overhead for small problems.

The results presented for the IBM PowerPC 604e in [28] show even more remarkable gains for the recursive algorithm without the optimized kernel. Here, the gain over LAPACK is around 10 to 30% for square matrices and is larger than that for tall, thin matrices. In extreme cases on the PowerPC 604e, we observe performance differences of up to a factor of 2.7 compared to LAPACK DGEQRF. The fact that the new algorithm gains more on the PowerPC 604e processor can be explained by that system's less efficient memory hierarchy, thereby making it more sensitive to memory reference patterns.

A parallel implementation on a four-way SMP PowerPC 604e system shows nearly ideal parallel speedups up to 1.97, 2.9, and 3.97 on two, three, and four processors, respectively [27, 28].

4.2. Solving Over- and Underdetermined Linear Systems. The algorithms of section 4.1 show how various level 2 and level 3 algorithms can be successfully replaced by recursive level 3 algorithms. The recursive techniques can also be applied to problems that normally are solved by a sequence of calls to different level 3 algorithms.

$X = \mathrm{RGELS}(A, B, nb)$
If $n \le nb$
  1. Factor $A = Q \left[\begin{smallmatrix} R \\ 0 \end{smallmatrix} \right]$; $\quad \tilde{B} \leftarrow Q^T B$; $\quad$ solve $\quad RX = \tilde{B}(1:n, :)$
else
  2. Let $A = \left[\begin{array}{cc} A_1 & A_2 \end{array} \right]$; $\quad B = \left[\begin{smallmatrix} B_1 \\ B_2 \end{smallmatrix} \right]$ with nb cols in A_1, nb rows in B_1
  3. Factor $A_1 = Q_1 \left[\begin{smallmatrix} R_{11} \\ 0 \end{smallmatrix} \right]$
  4. Set $\quad \left[\begin{smallmatrix} R_{12} & \tilde{B}_1 \\ A_{22} & \tilde{B}_2 \end{smallmatrix} \right] \leftarrow Q_1^T \left[\begin{array}{cc} A_2 & B \end{array} \right]$
  5. $X_2 = \mathrm{RGELS}(A_{22}, \tilde{B}_2, nb)$
  6. Solve $\quad R_{11} X_1 = \tilde{B}_1 - R_{12} X_2$; $\quad$ return $\quad X = \left[\begin{smallmatrix} X_1 \\ X_2 \end{smallmatrix} \right]$
endif

Fig. 4.2 *Recursive least squares RGELS algorithm for computing the solution to $AX = B$, where A is $m \times n$ $(m \ge n)$.*

Given an $m \times n$ matrix A with full row or column rank, and matrices X and B, both with *nrhs* columns, [29] considers the four problems solved by the LAPACK routine DGELS, namely, to compute the following:

1. linear least squares solution to $\min \|AX - B\|_F$ $(m \ge n)$;
2. linear least squares solution to $\min \|A^T X - B\|_F$ $(m < n)$;
3. minimum norm solution to $\min \|A^T X - B\|_F$ $(m \ge n)$;
4. minimum norm solution to $\min \|AX - B\|_F$ $(m < n)$.

In the LAPACK DGELS, these four problems are solved by first factorizing A into QR (or LQ). Then for the least squares solution, Q^T (or Q) is applied to B and a triangular system is solved. For the minimum norm solution, a triangular system is solved before Q (or Q^T) is applied to that solution.

The algorithms of [29] recursively reduce an over- or underdetermined problem to smaller and smaller sizes. Instead of, for example, first factorizing the complete matrix A, second updating the whole of B, and third performing a large triangular solve, these three operations are interleaved for each block of the matrix. This in turn leads to data reuse in the memory hierarchy among these operations. The QR factorization that is performed on block columns of A or A^T can be done using either a recursive or a standard algorithm, though we recommend the recursive algorithm for better performance.

In the following, we focus on problem 1 and only briefly comment on the other three problems. For further information, see [29].

Recursive Algorithm for Problem 1. The recursive algorithm for solving problem 1 is presented in Figure 4.2. It first partitions A in two parts over the column dimension and performs a QR factorization on A_1. Then A_2 and the whole of B is updated with respect to that factorization. A recursive call is made for solving a reduced-size least squares problem. Given that solution, a part of B is updated and the final part of the solution is computed. The stopping criteria is as follows: If A has no more than nb columns, then it is QR factorized and the corresponding part of the least squares problem is solved.

In practice, X overwrites the input B, Q is implicitly represented as $I - YTY^T$, and A is overwritten by R and Y. Similarly to the QR factorization, we reduce the amount of extra flops imposed by the compact WY representation for Q by splitting A with maximum nb columns in A_1.

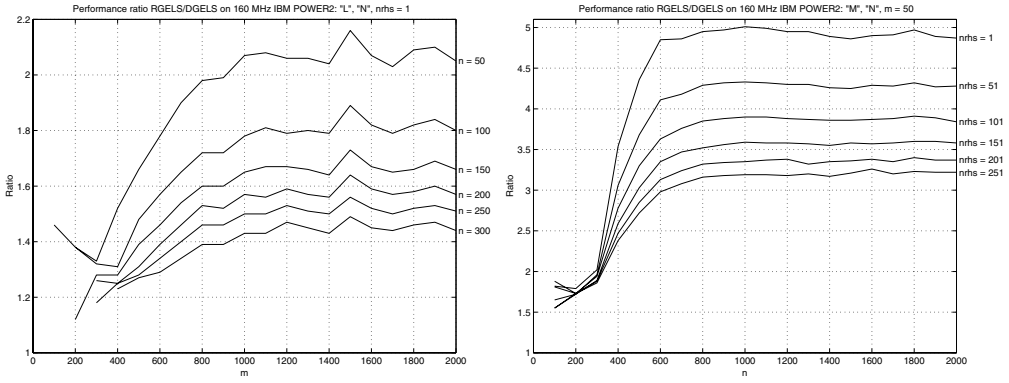

Fig. 4.3 *Performance ratio of RGELS to DGELS for computing the least squares solution with one right-hand side and varying m and n (left) and the minimum norm solution with $m = 50$ and varying n and number of right-hand sides (right) on the* 160 *MHz IBM Power2.*

Recursive Algorithms for Problems 2–4. The recursive algorithm for solving problem 3 is written in the same spirit. It also performs a recursion over the n-dimension, and for each recursive step it performs a QR factorization of a block of A, a triangular solve, updates with Q and Q^T, and a recursive call.

Problems 2 and 4 are traditionally, e.g., in LAPACK, solved using an LQ factorization of A. For A stored in column-major order, this implies that a Householder transformation is computed for each row of A. For each element, today's processors will transfer one cache line of elements (e.g., 16 elements), of which all but one are not immediately needed, nor can they be subsequently used if n is large. This causes severe cache thrashing and an approximate factor of two performance loss over using a QR approach.

The remedy is both simple and efficient. We perform an explicit transposition of A, turning problems 2 and 4 into problems 1 and 3, respectively, for which we already have efficient algorithms. This approach gives two advantages: the amount of code is reduced by a factor of two and the performance improvement over LAPACK is increased by an additional factor of two. In order to produce the same output as LAPACK DGELS, the factorized A^T is again transposed so that on output $A = LQ$.

Performance. An extensive performance evaluation for an early version of this routine is presented in [29]. It includes results for all four problems solved by RGELS. For each of these, results are presented for varying m, n, and *nrhs*. The new routine RGELS outperforms the LAPACK DGELS for all cases tested, and in extreme cases by up to a factor of five.

For illustration, we present some results in Figure 4.3, which show the performance ratio between RGELS and DGELS. The left-hand graph shows the performance for solving problem 1 with $nrhs = 1$ for varying m and n. The right-hand graph shows the performance for solving problem 4 with $m = 50$ for varying n and *nrhs*.

5. Recursive Blocked Solvers for Triangular Matrix Equations and Condition Estimation. The fourth case study concerns the successful application of recursive blocking to the solution of triangular matrix equations. In [60, 61, 62, 63], novel recursive blocked algorithms for solving one-sided and two-sided Sylvester-type matrix

Table 5.1 *One-sided (top) and two-sided (bottom) matrix equations.*

Name	Matrix equation	Acronym
Standard Sylvester (CT)	$AX - XB = C$	SYCT
Standard Lyapunov (CT)	$AX + XA^T = C$	LYCT
Generalized coupled Sylvester	$(AX - YB, DX - YE) = (C, F)$	GCSY
Standard Sylvester (DT)	$AXB^T - X = C$	SYDT
Standard Lyapunov (DT)	$AXA^T - X = C$	LYDT
Generalized Sylvester	$AXB^T - CXD^T = E$	GSYL
Generalized Lyapunov (CT)	$AXE^T + EXA^T = C$	GLYCT
Generalized Lyapunov (DT)	$AXA^T - EXE^T = C$	GLYDT

equations are presented. In this section, recursive blocking is combined with matrices stored in standard data format.

The classification in one-sided and two-sided matrix equations distinguishes the type of matrix product terms that appear and the way updates are performed in the recursive blocked algorithms. *One-sided* matrix equations include terms where the solution is only involved in matrix products of two matrices, e.g., $\mathrm{op}(A)X$ or $X\mathrm{op}(A)$, where $\mathrm{op}(A)$ can be A or A^T. Examples include the continuous-time standard Sylvester and Lyapunov equations, and the generalized coupled Sylvester (GCSY) equation. *Two-sided* matrix equations include matrix product terms of type $\mathrm{op}(A)X\mathrm{op}(B)$, and examples are the discrete-time standard and generalized Sylvester and Lyapunov equations. Table 5.1 gives a summary of the one-sided and two-sided matrix equations considered in [62, 63] together with their individual acronyms, where we use the abbreviations CT and DT for continuous-time and discrete-time, respectively. Triangular matrix equations appear naturally in different condition estimation problems for matrix equations and various eigenspace computations, and as reduced systems in standard algorithms. Related applications include block-diagonalization of matrices and matrix pairs [6, 19, 69, 4], computation of functions of matrices [62], the direct reordering of eigenvalues in the real (generalized) Schur form [4, 69], and the computation of additive decompositions of a (generalized) transfer function [71].

There is a quite extensive literature on the solution of matrix equations, and we refer to the following selection of fundamental papers [8, 38, 50, 16, 72, 37, 70, 79] that all present reliable algorithms. These standard methods are variants or generalizations of the Bartels–Stewart method [8]. The new recursive research focus is on the solution of the one-sided and two-sided triangular counterparts, which typically are obtained after an initial transformation of matrices (or regular matrix pairs) to Schur (or generalized Schur) form. Reliable and efficient algorithms for the reduction step can be found in LAPACK [4] and in [11, 12] for the standard case and [18] for the generalized case, where a blocked variant of the QZ method is presented.

In the recent new work, for each matrix equation, splittings are defined which in turn lead to a few smaller problems to be solved. These recursive splittings are applied to all "half-sized" triangular matrix equations and so on. We terminate the recursion when the new problem sizes (m and/or n) are smaller than a certain block size, $blksz$. Our approach guides us to choose $blksz$ such that at least a few submatrices involved in the current matrix equation fit in the first-level cache memory. However, in practice, we use a fixed $blksz = 4$, which in principle makes the implementations architecture-independent.

The recursive blocked algorithms allow sliding splittings, with the splitting points varying between the second and the penultimate rows and/or columns. By not split-

ting in the middle, the algorithms exhibit different memory access patterns and nonsquare updates, which in general degrade the performance. In the extreme cases we obtain the standard algorithms.

5.1. One-Sided and Coupled Sylvester-Type Matrix Equations. We start by reviewing recursive blocked algorithms for the real *continuous-time Sylvester* (SYCT) matrix equation:

$$AX - XB = C. \tag{5.1}$$

In (5.1), A has size $m \times m$, B has size $n \times n$, and both are upper triangular or upper quasi-triangular, i.e., in real Schur form. The right-hand side C and the solution X are of size $m \times n$ and, typically, the solution overwrites the right-hand side ($C \leftarrow X$). The SYCT equation (5.1) has a *unique solution* if and only if A and B have no eigenvalue in common or, equivalently, Sep[SYCT] $\neq 0$. (For a definition of the Sep-function see section 5.3.)

Depending on the sizes of m and n, three alternatives for doing a *recursive splitting* are considered. In Case 1 ($1 \leq n \leq m/2$), A is split by rows and columns, and C by rows only. Similarly, in Case 2 ($1 \leq m \leq n/2$), B is split by rows and columns, and C by columns only. Finally, in Case 3 ($n/2 < m < 2n$) both rows and columns of the matrices A, B, and C are split:

$$\begin{bmatrix} A_{11} & A_{12} \\ & A_{22} \end{bmatrix} \begin{bmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{bmatrix} - \begin{bmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}.$$

This recursive splitting results in the following four triangular SYCT equations:

$$\begin{aligned} A_{11}X_{11} - X_{11}B_{11} &= C_{11} - A_{12}X_{21}, \\ A_{11}X_{12} - X_{12}B_{22} &= C_{12} - A_{12}X_{22} + X_{11}B_{12}, \\ A_{22}X_{21} - X_{21}B_{11} &= C_{21}, \\ A_{22}X_{22} - X_{22}B_{22} &= C_{22} + X_{21}B_{12}. \end{aligned}$$

Conceptually, we start by solving for X_{21} in the third equation. After updating C_{11} and C_{22} with respect to X_{21}, one can solve for X_{11} and X_{22}. Both updates and the triangular Sylvester solves are independent operations and can be executed concurrently. Finally, one updates C_{12} with respect to X_{11} and X_{22} and solves for X_{12}. In practice, all four subsystems are solved using the recursive blocked algorithm.

Note that the three cases above are straightforward generalizations of the cases considered for the TRSM operation in section 2.1. However, for SYCT all three cases have a critical path at the block level that controls the ordering of the computations. If a splitting point ($m/2$ or $n/2$) appears at a 2×2 diagonal block, the matrices are split just below this diagonal block. We remark that the issue of 2×2 diagonal blocks corresponding to conjugate eigenvalue pairs would infer extra overhead if the a recursive data layout for matrices is used, and therefore a standard data layout is to be preferred [59].

In the discussion above, we have assumed that both A and B are upper triangular (or quasi-triangular). However, it is straightforward to derive similar recursive splittings for the triangular SYCT, where each of A and B can be in either upper or lower Schur form.

Similar recursive splittings for the triangular LYCT and GCSY equations are presented in [62], where important implementation issues are also discussed. These

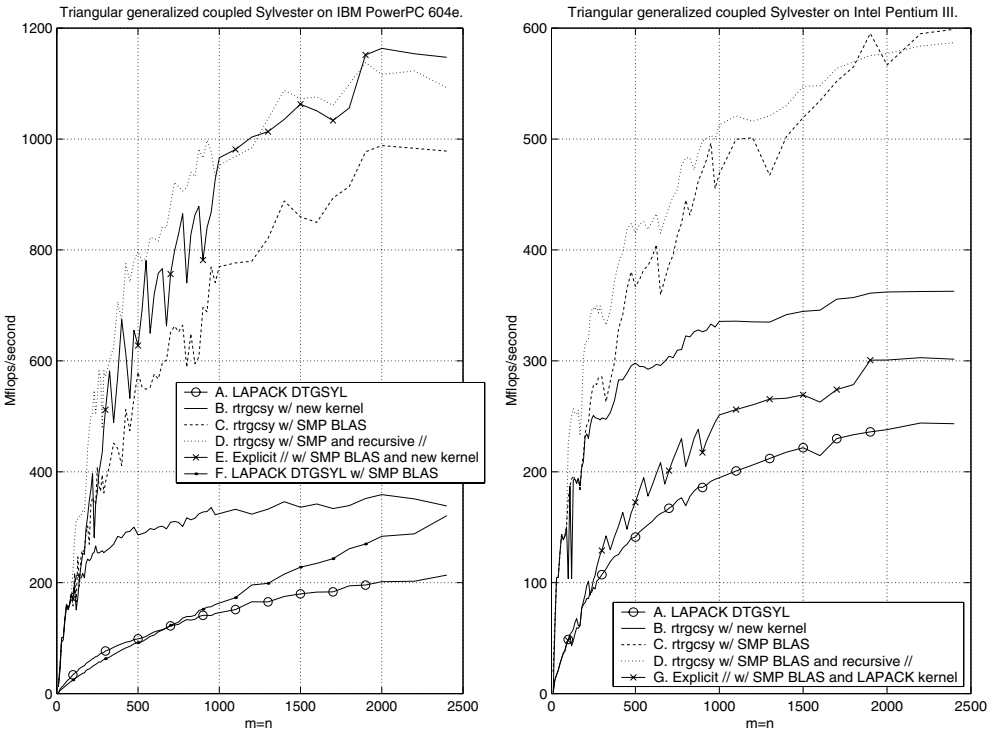

Fig. 5.1 *Performance results for the generalized coupled Sylvester equation ($m = n$) on an IBM PowerPC* 604*e (left) and an Intel Pentium* III *(right).*

include the impact of kernel solvers to the overall performance of the matrix equation solver, the design of superscalar kernels for solving small-sized matrix equations, and when to terminate the recursion. Moreover, different aspects regarding the choice of BLAS implementations and how to implement shared memory parallelism are discussed.

Due to the "one-sidedness" of the matrix equations, all updates with respect to the solution of subproblems in the recursion(s) are GEMM operations $C \leftarrow \beta C + \alpha\mathrm{op}(A)\mathrm{op}(B)$. Using the GEMM-based approach, some of them can be reorganized in efficient symmetric rank-$2k$ (SYR2K) operations [22, 21, 66, 67]. For details see [62].

Performance. This subsection ends by presenting some performance results for the GCSY equation. The choice of the GCSY equation was predicated since a good implementation of a level 3 explicit blocked algorithm exists [70, 69] and thereby makes the comparison as challenging and demanding as possible. In Figure 5.1, performance graphs for different algorithms and implementations, executed on IBM PowerPC 604e and Intel Pentium III processor–based systems, are displayed [62]. In total, seven different implementations are compared. Two graphs are the LAPACK DTGSYL (A) and a parallel variant which uses SMP BLAS (F). The LAPACK DTGSYL is an explicitly blocked level 3 algorithm based on a generalization of the Bartels–Stewart algorithm [70, 72]. Three graphs are the sequential recursive blocked *rtrgcsy* (B) and two parallel variants (C and D). C utilizes implicit data parallelism in the GEMM-

updates by linking to an SMP-aware implementation of BLAS. The routine D also solves subsystems concurrently and thereby makes use of task parallelism in the recursion tree. The last two (E and G) are new explicitly parallelized implementations of a standard blocked method [70, 81, 80].

Since the LAPACK DTGSYL is mainly a level 3 routine, its performance increases with increasing problem sizes and levels out because only one level of blocking is employed. Nonetheless *rtrgcsy* shows over a 5-fold speedup with respect to LAPACK DTGSYL and an additional speedup up to 3.2 on a four processor PowerPC 604e node for large enough problems. The corresponding results on a two-processor Intel Pentium III show up to 2.6-fold speedup with respect to LAPACK DTRSYL and additionally up to an 1.6-fold speedup on two processors. Comparisons have also been done with rectangular matrices. For the case $m = 10n$, the recursive implementations are twice as fast as LAPACK for problem size $(m, n) = (1000, 100)$. By using SMP versions, an extra speedup of 2.5 is achieved on the IBM PowerPC 604e, where for larger problems even better results occur [62].

For additional results and discussion we refer to [62]. Notably, the recursive blocked implementation *rtrsyct* shows between a 2-fold to a 35-fold speedup with respect to LAPACK DTRSYL and an additional speedup up to 2.8 on a four processor PowerPC 604e node for large enough problems. The LAPACK DTRSYL implements an explicit Bartels–Stewart solver and is mainly a level 2 routine, which explains its relatively poor performance behavior.

5.2. Two-Sided and Generalized Sylvester and Lyapunov Matrix Equations. Some of the matrix equations in Figure 5.1 can be seen as special cases of other formulations. In practice, this study treats all matrix equations separately, since either these equivalences include matrix inversion (when transforming a generalized matrix equation to a standard counterpart) or the matrix equations have symmetry structure that we want to take advantage of in the algorithms.

To illustrate such a case consider the real *generalized discrete-time Lyapunov* (GLYDT) matrix equation

$$AXA^T - EXE^T = C, \tag{5.2}$$

where A and E of size $n \times n$ are upper quasi-triangular and upper triangular, respectively. In other words, (A, E) is in generalized Schur form. If C is symmetric, then X is symmetric as well. Mathematically, GLYDT is as a special case of GSYL (with $A = B$ and $C = D$).

The GLYDT equation (5.2) has a *unique symmetric solution* if and only if $C = C^T$ and the eigenvalues λ_i of $A - \lambda E$ satisfy $\lambda_i \lambda_j \neq 1$ for all i and j (with the convention that $0 \cdot \infty = 1$), or, equivalently, Sep[GLYDT] $\neq 0$. (For a definition of the Sep-function see section 5.3.) If C is (semi)definite and $|\lambda_i(A - \lambda E)| < 1$ for all i, then a unique (semi)definite solution exists [79].

Since all matrices are square, the *recursive splitting* is done on both the rows and columns of A, E, and C [63]:

$$\begin{bmatrix} A_{11} & A_{12} \\ & A_{22} \end{bmatrix} \begin{bmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{bmatrix} \begin{bmatrix} A_{11}^T & \\ A_{12}^T & A_{22}^T \end{bmatrix} - \begin{bmatrix} E_{11} & E_{12} \\ & E_{22} \end{bmatrix} \begin{bmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{bmatrix} \begin{bmatrix} E_{11}^T & \\ E_{12}^T & E_{22}^T \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}.$$

Since $X_{21} = X_{12}^T$, the recursive splitting results in three triangular GLYDT equations:

$$\begin{aligned}
A_{11}X_{11}A_{11}^T - E_{11}X_{11}E_{11}^T &= C_{11} - A_{12}X_{12}^TA_{11}^T - (A_{11}X_{12} + A_{12}X_{22})A_{12}^T \\
&\quad + E_{12}X_{12}^TE_{11}^T + (E_{11}X_{12} + E_{12}X_{22})E_{12}^T, \\
A_{11}X_{12}A_{22}^T - E_{11}X_{12}E_{22}^T &= C_{12} - A_{12}X_{22}A_{22}^T + E_{12}X_{22}E_{22}^T, \\
A_{22}X_{22}A_{22}^T - E_{22}X_{22}E_{22}^T &= C_{22}.
\end{aligned}$$

We start by solving for X_{22} in the third equation. After updating C_{12} with respect to X_{22}, we can solve for X_{12}. Finally, after updating C_{11} with respect to X_{12} and X_{22}, we solve for X_{11}.

Four of the two-sided matrix product updates of C_{11} can be expressed as two SYR2K operations,

$$\begin{aligned}
C_{11} &= C_{11} - (A_{11}X_{12})A_{12}^T - A_{12}(A_{11}X_{12})^T, \\
C_{11} &= C_{11} + (E_{11}X_{12})E_{12}^T + E_{12}(E_{11}X_{12})^T,
\end{aligned}$$

where $A_{11}X_{12}$ and $E_{11}X_{12}$ are triangular matrix multiply (TRMM) operations. The GEMM-rich updates

$$C_{11} = C_{11} - A_{12}X_{22}A_{12}^T \quad \text{and} \quad C_{11} = C_{11} + E_{12}X_{22}E_{12}^T$$

with $C_{11} = C_{11}^T$ and $X_{22} = X_{22}^T$ can be performed efficiently (see the SLICOT MB01RD subroutine [90]).

Performance. In contrast to the one-sided matrix equations, the recursive blocked algorithms for the two-sided matrix equations require extra workspace and execute more flops compared to the standard elementwise algorithms. The extra workspace is needed in the evaluation of the two-sided matrix multiplications of type $\mathrm{op}(A)X\mathrm{op}(B)$. Despite the quite large flops penalties of the recursive blocked algorithms [63], they outperform the standard algorithms for large enough problems (obtaining tenfold speedups and more). For example, solving discrete-time Sylvester and Lyapunov equations with coefficient matrices of size 2000×2000 takes around one hour using the current routines in the SLICOT library [90], while the corresponding solution times of the new recursive blocked algorithms are less than one minute. This fact is mainly due to the difference in their data reference patterns, i.e., the order in which they access data and how many times the data is moved in the memory hierarchy of the target computer system. Clearly, the cost of redundant memory transfers can be devastating to the algorithm performance.

As for the one-sided matrix equations [62], we have developed new high-performance superscalar kernels for solving the remaining small-sized triangular matrix equations and lightweight GEMM operations, which implies that a larger part of the total execution time is spent in higher performing GEMM operations. Also for the two-sided matrix equations one can terminate the recursion with $blksz = 4$ without degrading performance. Moreover, recursive blocking allows the development of optimized two-sided matrix product kernels that take any symmetry properties as well as any triangular or trapezoidal structure of the matrices into account. These efficiencies allow one to maximize the performance of these two-sided matrix product operations, e.g., AXB^T. See [63] for more information about implementation issues and performance results of the two-sided recursive blocked algorithms.

Table 5.2 *Sep-functions associated with one-sided (top) and two-sided (bottom) matrix equations.*

Z-matrix	Sep-function $= \sigma_{\min}(Z_{\_})$
$Z_{\mathrm{SYCT}} = I_n \otimes A - B^T \otimes I_m$	$\inf_{\|X\|_F=1} \|AX - XB\|_F$
$Z_{\mathrm{LYCT}} = I_n \otimes A + A \otimes I_n$	$\inf_{\|X\|_F=1} \|AX - X(-A^T)\|_F$
$Z_{\mathrm{GCSY}} = \begin{bmatrix} I_n \otimes A & -B^T \otimes I_m \\ I_n \otimes D & -E^T \otimes I_m \end{bmatrix}$	$\inf_{\|(X,Y)\|_F=1} \|(AX - YB, DX - YE)\|_F$
$Z_{\mathrm{SYDT}} = B \otimes A - I_n \otimes I_m$	$\inf_{\|X\|_F=1} \|AXB^T - X\|_F$
$Z_{\mathrm{LYDT}} = A \otimes A - I_n \otimes I_n$	$\inf_{\|X\|_F=1} \|AXA^T - X\|_F$
$Z_{\mathrm{GSYL}} = B \otimes A - D \otimes C$	$\inf_{\|X\|_F=1} \|AXB^T - CXD^T\|_F$
$Z_{\mathrm{GLYCT}} = E \otimes A + A \otimes E$	$\inf_{\|X\|_F=1} \|AXE^T - EX(-A^T)\|_F$
$Z_{\mathrm{GLYDT}} = A \otimes A - E \otimes E$	$\inf_{\|X\|_F=1} \|AXA^T - EXE^T\|_F$

5.3. Matrix Equation Solvers in Condition Estimation. All linear matrix equations can be written as a linear system of equations $Zx = c$, where Z is a *Kronecker product matrix representation* of the associated Sylvester-type operator, and the solution x and the right-hand side c are represented in vec$(\cdot)$ notation. vec(X) denotes a column vector with the columns of X stacked on top of each other. The Z-matrices associated with the matrix equations of Table 5.1 are listed in Table 5.2 (left part).

An important quantity in the perturbation theory for Sylvester-type equations is the *separation between two matrices* [93], defined as

$$\mathrm{Sep}[A, B] = \inf_{\|X\|_F=1} \|AX - XB\|_F = \sigma_{\min}(Z_{\mathrm{SYCT}}),$$

where $\sigma_{\min}(Z_{\mathrm{SYCT}}) \geq 0$ is the smallest singular value of Z_{SYCT}. A review of some of its characteristics is in order: $\mathrm{Sep}[A, B] = 0$ if and only if A and B have a common eigenvalue; $\mathrm{Sep}[A, B]$ is small if there is small perturbation of A or B that makes them have a common eigenvalue. The Sep-function may be much smaller than the minimum distance between the eigenvalues of A and B. The Sep-functions associated with the matrix equations of Table 5.1 are listed in Table 5.2 (right part), which all can be expressed as $\sigma_{\min}(Z_{\_})$, where $Z_{\_}$ corresponds to the Kronecker product matrix representation of the associated Sylvester-type operator. Indeed, the matrix equations considered have a unique solution if and only if the associated Sep-functions are nonzero, i.e., $\sigma_{\min}(Z_{\_}) > 0$.

Computing $\sigma_{\min}(Z_{\mathrm{SYCT}})$ is typically an $O(m^3n^3)$ operation, which is impractical already for moderate values on m and n. In [68], it is shown how reliable $\mathrm{Sep}[\mathrm{SYCT}]^{-1}$-estimates can be computed to the cost $O(mn^2 + m^2n)$ by solving triangular matrix equations:

$$\frac{\|x\|_2}{\|c\|_2} = \frac{\|X\|_F}{\|C\|_F} \leq \|Z_{\mathrm{SYCT}}^{-1}\|_2 = \frac{1}{\sigma_{\min}(Z_{\mathrm{SYCT}})} = \mathrm{Sep}^{-1}.$$

The right-hand side C is chosen such that the lower bound gets as large as possible. This leads to a Frobenius-norm–based estimate. For computation of 1-norm–based estimates see [49, 52, 68]. Reliable estimates of Sep[GCSY] (the separation between two matrix pairs [93]) are presented and discussed in [70, 69, 72]. The same techniques can be used for estimating all the Sep-functions of Table 5.2. The underlying perturbation theory for these Sylvester-type equations is presented in [53, 65, 54].

Performance for Unreduced Matrix Equations with Optional Condition Estimation. We end this subsection by presenting some performance results that show

Table 5.3 *Timings for solving unreduced two-sided matrix equations (GLYDT) with optional condition estimation. (*Job $= X$*, compute solution only;* Job $= X +$ Sep*, compute solution and Sep-estimation.) Results from* 375 *MHz IBM Power3.*

(a)

	SG03AD using SG03AX		SG03AD using *rtrglydt*			
n	Total time	Solver part	Total time	Solver part	Speedup	Job
50	0.0277	49.9 %	0.0185	20.1 %	1.50	X
100	0.180	51.2 %	0.0967	9.0 %	1.86	X
250	2.89	46.8 %	1.62	4.7 %	1.79	X
500	59.0	42.3 %	34.5	1.5 %	1.71	X
750	303.4	42.0 %	177.5	0.9 %	1.71	X
1000	646.6	44.6 %	361.8	1.0 %	1.79	X
50	0.117	87.6 %	0.0263	45.6 %	4.44	X+Sep
100	0.709	87.3 %	0.152	40.6 %	4.68	X+Sep
250	9.98	84.5 %	2.08	25.4 %	4.81	X+Sep
500	178.6	80.9 %	37.8	9.4 %	4.73	X+Sep
750	924.1	80.9 %	184.4	4.5 %	5.01	X+Sep
1000	2076.6	82.7 %	391.8	8.4 %	5.30	X+Sep

(b)

	SG03AD using SG03AX		SG03AD using *rtrglydt*			
n	Total time	Solver part	Total time	Solver part	Speedup	Job
50	0.0306	44.7 %	0.019	11.2 %	1.61	X
100	0.213	43.2 %	0.129	6.5 %	1.65	X
250	3.18	42.5 %	1.90	3.9 %	1.67	X
500	41.8	59.7 %	17.3	2.9 %	2.41	X
750	187.1	68.2 %	61.1	2.7 %	3.06	X
1000	428.9	67.2 %	144.1	2.5 %	2.98	X
50	0.120	85.4 %	0.0285	39.1 %	4.19	X+Sep
100	0.741	83.5 %	0.166	26.2 %	4.47	X+Sep
250	10.3	82.1 %	2.69	31.0 %	3.83	X+Sep
500	161.3	89.5 %	20.0	15.3 %	8.06	X+Sep
750	807.7	92.6 %	67.9	12.1 %	11.89	X+Sep
1000	1857.4	92.4 %	174.0	18.9 %	10.68	X+Sep

the impact of the choice of triangular matrix equation solver on both the time to solve unreduced matrix equations and the time to compute an estimate of the associated Sep-function. Although the transformation of an unreduced matrix equation to a triangular counterpart and the backtransformation of the solution are normally at least as costly operations (measured in flops) as the triangular solve, the impact of using our recursive triangular solvers can be substantial.

For illustration, the SLICOT routine SG03AD which solves unreduced GLYDT equations with an option to compute an estimate of the separation Sep[GLYDT] $= \sigma_{\min}(Z_{\mathrm{GLYDT}})$ can be used. In Table 5.3a, timings for the SG03AD routine are displayed for problem sizes ranging from 50 to 1000 using two different triangular matrix equation solvers [63]. These solvers are called SG03AX in SLICOT; they implement a variant of the Bartels–Stewart method by calling BLAS [79], and the recursive blocked *rtrglydt* algorithm [63]. In the second column, the total times for solving an unreduced system with SG03AX as the triangular solver are displayed. This includes the time for the generalized Schur factorization and the backtransformation of the solution. In the fourth and fifth columns, similar results are displayed when SG03AX is replaced by the *rtrglydt* routine. We see up to a factor 1.9 speedup for the problem sizes considered. The numbers in the lower part of Table 5.3a also include the time for computing a 1-norm–based estimate for Sep[GLYDT]. The condition

$$A \sim \begin{pmatrix} 1 & 11 & 21 & 31 & 41 & 51 & 61 \\ 2 & 12 & 22 & 32 & 42 & 52 & 62 \\ 3 & 13 & 23 & 33 & 43 & 53 & 63 \\ 4 & 14 & 24 & 34 & 44 & 54 & 64 \\ 5 & 15 & 25 & 35 & 45 & 55 & 65 \\ 6 & 16 & 26 & 36 & 46 & 56 & 66 \\ 7 & 17 & 27 & 37 & 47 & 57 & 67 \\ 8 & 18 & 28 & 38 & 48 & 58 & 68 \\ 9 & 19 & 29 & 39 & 49 & 59 & 69 \\ 10 & 20 & 30 & 40 & 50 & 60 & 70 \end{pmatrix} \sim \left(\begin{array}{cccc|cccc} 1 & 5 & 9 & 13 & 49 & 53 & 57 & * \\ 2 & 6 & 10 & 14 & 50 & 54 & 58 & * \\ 3 & 7 & 11 & 15 & 51 & 55 & 59 & * \\ 4 & 8 & 12 & 16 & 52 & 56 & 60 & * \\ \hline 17 & 21 & 25 & 29 & 65 & 69 & 73 & * \\ 18 & 22 & 26 & 30 & 66 & 70 & 74 & * \\ 19 & 23 & 27 & 31 & 67 & 71 & 75 & * \\ 20 & 24 & 28 & 32 & 68 & 72 & 76 & * \\ \hline 33 & 37 & 41 & 45 & 81 & 85 & 89 & * \\ 34 & 38 & 42 & 46 & 82 & 86 & 90 & * \\ * & * & * & * & * & * & * & * \\ * & * & * & * & * & * & * & * \end{array}\right)_{(BC)}$$

Fig. 6.1 *Square block column (BC) data format for matrix A of size 10×7, $blksz = 4$. The number in location (i, j) is $a(i, j)$'s storage position in A.*

estimation process includes repeated calls (typically five) to the generalized triangular solver, and as expected we see a four- to fivefold speedup when using the new recursive solver.

In Table 5.3b [63], the corresponding results are displayed when we have replaced the LAPACK routines DGGHRD and DHGEQZ by Dackland–Kågström's blocked Hessenberg-triangular reduction and QZ algorithms [18] for transforming the regular pair (A, E) to generalized Schur form. For large enough problems, these algorithms give another factor of two speedup.

6. Blocked Standard Algorithms and Hybrid Data Formats. The previous sections have shown the generality and the strength of the recursive approach for developing efficient and robust algorithms and library software for dense linear algebra computations. However, other techniques that follow the algorithms and architecture approach can also be very useful. By changing the data format, and by implementing kernel routines which are optimal when using this new data format, very high performance can be achieved (see section 2). This section briefly illustrates the use of square blocked data formats combined with standard (nonrecursive) algorithms.

Matrices stored in block format are split in rectangular or square blocks, where each block is stored either in row or column order. In Figure 6.1, the layout of individual elements in memory is displayed when padding is used for the last block row and column.

Now, to obtain a blocked standard algorithm the scalar operations in a pointwise algorithm are replaced by calls to kernel routines for the submatrices. There are several ways to store and reference the blocks. One is to use pointers to reference blocks [75]. Another is to use three- or four-dimensional arrays [45, 44]. The latter provides a straightforward way to address the blocks in Fortran programs, as easy as addressing scalars in textbook elementwise algorithms. When the scalar algorithm uses a fused multiply and add, the block algorithms use a GEMM kernel. Similarly, a scalar division or multiply translates into a TRSM or TRMM kernel, respectively.

We remark that our hybrid data formats, in general, eliminate excessive data copying. Such copyings are necessitated when TLB (translation look-aside buffer, contains cross-references between the virtual and real addresses) and/or cache utilization is poor due to bad leading dimensions and large access strides in the operands. The copying is eliminated by use of the hybrid data formats, as the blocks of the operands are already contiguous and fit into L1 cache.

The standard or packed blocked formats are also applicable to triangular arrays. The main benefit of these formats is that they also allow for level 3 performance while

using about half the storage of the full array cases (see also section 3). Implementation details and performance results are given in [45, 44]. The good performance of the square blocked algorithms is yet another example of the importance of matching the algorithm with the data format. However, the square blocked algorithms give only one level of cache blocking. For some processors, this can be enough, though. Even so, it is possible to explicitly block on top of a square block format for higher levels of the memory hierarchy. On the other hand, the recursive blocked algorithms provide the potential to automatically block for all levels of cache, which is especially important for architectures with an unbalanced memory system hierarchy.

7. Related and Complementary Work. There are several other groups that have been contributing to this new research area. In the following, but without pretending to be complete, we give a short overview of some of these contributions that either relate to or complement the case studies presented.

7.1. Recursive Algorithms and Hybrid Data Structures. We start by discussing some related work on recursive algorithms and hybrid data structures. Most work concern matrix multiplication, which is a very regular operation with nice properties. Also, it is the natural algorithm to look at. For example, the resulting GEMM suboperations after a block partitioning of the matrices involved can be executed in any order. This fact makes it possible to formulate several different recursive blocked variants of the standard algorithm (see section 2.1). Moreover, we also have the recursive algorithms of Winograd and Strassen [94, 26, 20, 56]. Nevertheless, the performance of the GEMM operation can be highly sensitive to memory system behavior.

Recently, Chatterjee et al. [15] evaluated five recursive data layouts for two-dimensional arrays. Four of these are different Morton variants (Z, U, X, G) and the last one is the Hilbert layout for a space-filling curve [84]. These recursive layouts are used in three recursive blocked algorithms for matrix multiplication (standard, Strassen, Winograd). One of their conclusions is that recursive array layouts significantly outperform traditional array layouts for the blocked recursive standard algorithm, where splitting is used in all three problem dimensions (see section 2), while they offer little improvement for the block versions of the Strassen and Winograd algorithms. Another contribution of their work is explicit expressions for the recursive layout functions. Moreover, they claim that the overheads due to addressing these recursive layouts are only marginal.

Valsalam and Skjellum [97] presented a conceptual framework for developing high-performance polyalgorithmic matrix multiplication routines using hierarchical storage formats and optimized processor-specific kernels. The hierarchical matrix format consists of four levels, and their motivation for proposing such a complex data format was to efficiently apply the Z-Morton ordering to arbitrary-sized matrices. The polyalgorithms are based on comparative evaluations of different algorithms (standard (iterative); modified recursive (Frens and Wise [102]); oscillating iterative—incorporates the two-miss feature of the modified recursive algorithm—always possible to compute the next block product by reusing one of the three blocks and loading the other two [102]; and Strassen's algorithm). They show very good performance results on several platforms. One important factor is the use of lightweight interfaces for linear algebra kernels, which operate on blocks that fit in the L1 cache.

In [95], Toledo used a recursive approach for the LU factorization with partial pivoting and analyzed the locality of reference in the new algorithm, showing that the recursive blocked algorithm experiences fewer cache misses than the right-looking algorithm.

Rabani and Toledo used our recursive QR factorization algorithm in out-of-core implementations of the QR factorization and the singular value decomposition (SVD) in the SOLAR software library [82]. The recursive QR factorization gives excellent performance for very tall, thin matrices, too large to be stored in memory. If A is $m \times n$ with $m > n$, the SVD is computed by an out-of-core factorization $A = QR$ followed by an in-core SVD of the (small) matrix R. In [83], these algorithms were used in an orthogonalization step of an out-of-core filter diagonalization method for electronic structure calculations. This work clearly emphasizes the recursive algorithms ability to perform an efficient blocking for an arbitrary number of levels of the memory hierarchy.

More recently, Irony, Shklarski, and Toledo also applied these techniques to develop recursive multifrontal supernodal algorithms for sparse Cholesky factorization [58]. They used the recursive algorithm both on the complete sparse problem and on each dense matrix block. Moreover, they used our recursive blocked data structures described in section 2.2.1 for matrix storage. Their algorithm outperformed the SGI SCSL library software as the matrix size exceeds the L2 cache memory.

Frens and Wise [32] presented a Givens-based recursive QR factorization for quadtree matrices in Morton-order storage. The disadvantage of Givens rotations is that this algorithm requires significantly more flops than a Householder-based counterpart, partially offsetting the increased efficiency gained by recursive blocking.

Also Dongarra, Eijkhout, and Łuszczek adopted the recursive technique to sparse matrices, in this case sparse LU factorization without pivoting [24]. Since no pivoting is performed, the matrix partitioning can be performed over both dimensions and the recursive factorization call is made for the top-left block. Besides that, the main differences from the corresponding dense algorithm are how the matrix is stored and that calls are made to sparse BLAS instead of dense BLAS. The matrix is stored in a hybrid recursive format. In summary, this study shows potential for recursion also for sparse matrix factorizations, even though there is still a need for further algorithmic improvements and understanding of when to use these algorithms.

7.2. Automated Generation of Library Software and Compiler Technology. A complementary direction of research is the development of tools for automatic generation of software optimized for a given architecture. The main motivation is to be able to provide tuned library software along with the rapid development of improved and new increasingly powerful computer systems. The knowledge and experience gained from the research on algorithms, new data structures, and library software is a mandatory prerequisite for successful research in this area. We briefly discuss some recent work based on empirical techniques and heuristic optimization as well as some compiler technology work.

The ATLAS (Automatically Tuned Linear Algebra Software) project applies empirical techniques in order to provide portable performance [100, 5]. Typically, a code generator is used for the GEMM kernel $C = C - A^T B$ for matrices usually stored in square blocked format to provide the many different ways of performing this given operation, and search scripts and timings are used to find the best way to "unroll" for a given architecture. The first project in this vein is PHiPAC [9]. Today, ATLAS provides support for the level 3 BLAS, most of level 1–2 BLAS, and some factorization routines. Their current implementation of the level 3 BLAS is a recursive GEMM-based design, which builds on the GEMM-based level 3 BLAS by Kågström, Ling, and Van Loan [66, 67] and our blocked recursive approach [46, 47].

ATLAS reports impressive performance results on a broad range of state-of-the-art processors [5], although not for all. Certainly, there are instances when one does better by using analytical tools and techniques (e.g., see [67, 97, 40]) and special architectural features in developing efficient library software. Today, ATLAS is the most widely used software that provides a quick-to-implement and tuned level 3 BLAS for several different architectures. This provides quick and easy access to high-performance libraries such as LAPACK [4], SLICOT [90], etc. In this context, we remark that the current LAPACK is heavily based on calls to BLAS libraries with repetitive data copying. Today's superscalar processors with deep memory hierarchies tend increasingly to require optimized kernel routines with light interfaces, with data structures requiring minimal or no data copy, as well as exploit new architectural features such as the Intel Streaming SIMD Extensions (SSE and SSE2). Additional related recent work by the UC Berkeley group includes statistical models for automatic performance tuning and memory hierarchy optimization for sparse matrix kernels [98, 99].

The FLAME (Formal Linear Algebra Methods Environment) project [41, 42] develops a framework that facilitates the derivation and implementation of linear algebra algorithms on high-performance computer systems. A formal technique is applied to systematically derive well-known algorithms and their variants for a given matrix operation. So far, the framework has been demonstrated on a few case studies, including LU factorization. Results are presented that show performance comparable to an ATLAS-generated high-performance LU implementation. FLAME is an interesting concept based on the derivation of loop invariants for iterative algorithm implementations. Although FLAME can derive algorithms, there is of course no guarantee that these are numerically stable. We remark that (semi)automatic generation of software should be based only on algorithms with proven good stability characteristics.

Park, Hong, and Prasanna [77] have studied recursive and block standard data layouts. They call the latter BDL (block data layout) and our recursive layout RBR is called Morton layout of the blocks. They perform an experiment of passing through a large $n \times n$ matrix in both the row and column directions. They show that L1 and L2 cache misses and TLB misses are minimized for any data layout for certain block sizes when using these data layouts. Their result provides both analytic and experimental evidence of our heuristic and experimental claims about the optimality of our data layouts.

From a compiler technology point of view, the ultimate goal is to be able to take an arbitrary code as input and generate optimal tuned code as output for a given language and computer system. Locality of reference issues such as blockability have been studied by several compiler groups during the last decade. The first is a prize-winning paper by Wolf and Lam [103]. This work applies to loop nests not possessing complicated data-dependent branches. The paper looks at matrix multiplication, SOR, and LU without pivoting. They present a loop transformation algorithm based on two concepts: a mathematical formulation of reuse and locality, and a loop transformation theory that unifies the various transforms as unimodular matrix transformations. For $n = 500$ matrix multiplication they get a 2.75 times performance boost using both cache and register tiling. And for eight processors of an SGI 4D/380 using 64×64 cache blocks and 4×2 register blocks they get speedups of over seven. The best result for all eight processors was about 64 Mflops/s. For $n = 500$ LU without pivoting 32×32 cache blocks (no register tiling) was best. However, the best result, again with all eight processors, was 25 Mflops/s.

The first ESSL release for RISC workstations [55] written in Fortran featured cache and TLB blocking and data copying to produce level 3 BLAS and dense linear algebra software that achieved about 90% of the theoretical peak performance of the IBM Power1 for almost all matrix sizes. Similar results were modeled and experimentally verified in [103] and [73].

Recent work on compiler blockability of dense matrix factorizations by Carr and Lehoucq [14] used a GEMM-based approach [66, 67]. Also, compiler groups at Cornell University and Rice University have started research on recursive blocked algorithms and data structures.

Ahmed and Pingali [2] described compiler technology to translate iterative algorithms for matrix multiplication and Cholesky factorization into block-recursive form. They also studied the cache behavior and performance of these compiler-generated recursive blocked codes. The results are promising but the automated generated codes have yet to reach the levels of hand-tuned versions.

In [104], Yi, Adve, and Kennedy presented work on compiler transformations that convert loop nests into recursive form. Compared to previous work on code transformations for improving data locality, this work has two main features. It combines the effect of blocking at multiple levels into one single transformation and, when applied to multiple loop nests, it unifies the blocking and loop fusion transformations.

In summary, these efforts of trying out the recursive approach in compiler technology for dense matrix computations have given positive results that motivate further studies. One challenge is the development of a linear algebra compiler.

8. Concluding Remarks. Recursion is a well-known concept and technique for formulating algorithms in computing science. This survey shows that recursive algorithms and related data structures apply well to the area of dense linear algebra. Recent results are novel variably blocked algorithms and hybrid data formats for the efficient solution of dense linear algebra problems on today's memory-tiered systems. Furthermore, this research has produced robust and efficient library software, which is publicly available. The accuracy of the results computed by our recursive implementations are overall very good, the same accuracy as obtained by corresponding LAPACK [4] and SLICOT [90] routines.

The new high-performance software implementations are based on data locality and superscalar optimization techniques. In summary, application of the recursive techniques has demonstrated how excellent performance can be obtained by using the first, the first two, or all three of the following components: recursive blocked algorithms; superscalar kernels; and hybrid data structures.

The recursive blocked algorithms improve on the temporal data locality (see sections 2, 3, 4, and 5); the hybrid data formats improve on the spatial data locality (see sections 2 and 3) to the extent that is possible (see [77]). Each block gives optimal temporal data locality when in L1 cache. With data copy, sometimes not necessary, spatial data locality becomes optimal with respect to cache and TLB misses when using a standard algorithm [77]. Moreover, portable and generic superscalar kernels ensure that all functional units on the processor(s) are used efficiently (see sections 2, 3, 4, and 5). For the routines where hybrid data formats are used some extra work to convert data from and to standard storage formats is required. The time needed for the $\mathcal{O}(mn)$ copy work of an $m \times n$ matrix is typically between 5 to 20% for problem sizes up to 100 and can be neglected for large-sized matrix computations. However, it is difficult to completely characterize and understand which gains contribute to better temporal and spatial localities, respectively, and more research on this issue

Table 8.1 *Sample performance results of LAPACK expert drivers with and without RECSY routines (375 MHz IBM Power3 platform).*

n	b	DTRSEN (sec)	recsyct / DTRSYL	Speedup	DTGSEN (sec)	recgcsy / DTGSYL	Speedup
100	10	6.19e-3	0.72	1.38	93.5e-3	0.53	1.89
100	50	18.4e-3	0.63	1.60	254.e-3	0.52	1.92
1000	10	2.03	0.30	3.33	4.44	0.60	1.66
1000	100	18.7	0.24	4.08	41.5	0.56	1.80
1000	500	81.7	0.12	8.38	135.7	0.49	2.05

is warranted. Nonetheless, the results from the four case studies and related work reviewed show the strength and potential of the recursive techniques presented and applied to dense linear algebra problems.

As mentioned, one can always get equally good results by explicit multilevel blocking. However, this is a much more error-prone and time-consuming process, and usually the recursive blocked approach leads to much simpler algorithms that at most require knowledge of the L1 cache characteristics. In combination with generic superscalar kernels, the algorithms and software are more portable.

Some software implementations are included in the IBM ESSL library [56]. For symmetric positive definite matrices in packed format, S/DPPF and S/DPPFCD compute the Cholesky and LDL^T factorizations, respectively. Based on these factorizations, S/DPPS solves a linear system and S/DPPICD computes the inverse, determinant, and the condition number for a given matrix. For general matrices in standard storage format, DGEQRF computes the QR factorization (also in parallel) and DGELS computes the least squares and minimum norm solutions to over- and underdetermined linear systems.

All software implementations of the triangular matrix equations are available in the RECSY library [64]. It comprises a set of Fortran 90 routines, which uses recursion and OpenMP for shared memory parallelism to solve eight different matrix equations, including continuous-time as well as discrete-time standard and generalized Sylvester and Lyapunov equations. Instead of using the native routines, the library also provides wrapper routines, which overload SLICOT [90] and LAPACK [4] routines. By linking with the library, calls to SLICOT and LAPACK Sylvester-type matrix equations solvers will be replaced by calls to the optimized RECSY equivalents.

As an illustration of this library performance, in Table 8.1, speedups a user is likely to see compared to LAPACK is given in a slightly larger context. Here, the LAPACK expert drivers DTRSEN and DTGSEN, which compute the Schur and generalized Schur forms, respectively, and the condition number of a selected invariant or deflating subspace are shown. They are called for different matrix sizes n and sizes b of the selected subspace. Besides condition estimation, the computations include reduction to (generalized) Schur form, reordering of eigenvalues such that the eigenvalues of the selected subspace appear in the $(1, 1)$ block of the matrix (DTRSEN) or matrix pencil (DTGSEN). It is only in the 1-norm condition estimation of the Sep-function that calls to the LAPACK triangular Sylvester-type solvers DTRSYL (mainly a level 2 implementation) and DTGSYL (a level 3 implementation [70, 69]) are replaced by calls to recsyct and recgcsy, respectively. For large enough n and b one will see a speedup of eight for calling the expert drivers DTRSEN and two for DTGSEN when using the new recursive matrix equation solvers. In conclusion, RECSY provides a significant improvement.

We end our survey by remarking that the successful application of recursive techniques to matrix computations leaves some open problems in how to apply recursion to improve performance of "two-sided" linear algebra operations, such as the reduction to (generalized) Hessenberg, symmetric tridiagonal, or bidiagonal forms. One challenge is to see if recursion can reduce the nontrivial fraction of level 1 and level 2 operations that are currently applied to both sides of the matrix under reduction.

Acknowledgments. This research was conducted using the resources of the High Performance Computing Center North (HPC2N), PDC–Parallelldatorcentrum at KTH, Stockholm, and UNI-C, Danish Technical University, Lyngby. Several people have been instrumental to our work. Especially, we thank Per Ling, former member of the Umeå team; the former Master students André Henriksson, Olov Gustavsson, and Andreas Lindkvist; and Bjarne Andersen and Jerzy Wasniewski of UNI-C, Lyngby, for their contributions. We are also grateful for constructive comments from James Demmel, Randy LeVeque, Margaret Wright, and an anonymous referee, who have improved on both the readability and the content of the final manuscript.

REFERENCES

[1] R. C. Agarwal, F. G. Gustavson, and M. Zubair, *Exploiting functional parallelism of POWER2 to design high-performance numerical algorithms*, IBM J. Res. Develop., 38 (1994), pp. 563–576.

[2] N. Ahmed and K. Pingali, *Automatic generation of block-recursive codes*, in Euro-Par 2000 Parallel Processing, A. Bode et al., eds., Lecture Notes in Comput. Sci. 1900, Springer-Verlag, New York, 2000, pp. 368–378.

[3] B. Andersen, F. Gustavson, and J. Waśniewski, *A recursive formulation of Cholesky factorization of a matrix in packed storage*, ACM Trans. Math. Software, 27 (2001), pp. 214–244.

[4] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, 3rd ed., SIAM, Philadelphia, 1999.

[5] ATLAS, *Automatically Tuned Linear Algebra Software*, http://math-atlas.sourceforge.net/.

[6] Z. Bai, J. Demmel, and A. McKenney, *On computing condition numbers for the nonsymmetric eigenproblem*, ACM Trans. Math. Software, 19 (1993), pp. 202–223.

[7] J. Barnes and P. Hut, *A hierarchical O(nlogn) force calculation algorithm*, Nature, 324 (1986), pp. 446–449.

[8] R. H. Bartels and G. W. Stewart, *Solution of the equation* $AX + XB = C$, Comm. Assoc. Comput. Mach., 15 (1972), pp. 820–826.

[9] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel, *Optimizing matrix multiply using PHiPAC: A portable high-performance ANSI C methodology*, in Proceedings of the International Conference on Supercomputing, Vienna, 1997, pp. 340–347.

[10] C. Bischof and C. Van Loan, *The WY representation for products of Householder matrices*, SIAM J. Sci. Statist. Comput., 8 (1987), pp. s2–s13.

[11] K. Braman, R. Byers, and R. Mathias, *The multishift QR algorithm. Part* I: *Maintaining well-focused shifts and level* 3 *performance*, SIAM J. Matrix Anal. Appl., 23 (2002), pp. 929–947.

[12] K. Braman, R. Byers, and R. Mathias, *The multishift QR algorithm. Part* II: *Aggressive early deflation*, SIAM J. Matrix Anal. Appl., 23 (2002), pp. 948–973.

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

[14] S. Carr and R. B. Lehoucq, *Compiler blockability of dense matrix factorizations*, ACM Trans. Math. Software, 23 (1997), pp. 336–361.

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

[16] K.-W. E. Chu, *The solution of the matrix equations* $AXB - CXD = E$ *and* $(YA - DZ, YC - BZ) = (E, F)$, Linear Algebra Appl., 93 (1987), pp. 93–105.

[17] J. J. M. Cuppen, *A divide and conquer method for the symmetric tridiagonal eigenproblem*, Numer. Math., 36 (1981), pp. 177–195.
[18] K. Dackland and B. Kågström, *Blocked algorithms and software for reduction of a regular matrix pair to generalized Schur form*, ACM Trans. Math. Software, 25 (1999), pp. 425–454.
[19] J. Demmel and B. Kågström, *Computing stable eigendecompositions of matrix pencils*, Linear Algebra Appl., 88/89 (1987), pp. 139–186.
[20] J. W. Demmel and N. J. Higham, *Stability of block algorithms with fast level-3 BLAS*, ACM Trans. Math. Software, 18 (1992), pp. 274–291.
[21] J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling, *Algorithm* 679: *A set of level* 3 *basic linear algebra subprograms*, ACM Trans. Math. Software, 16 (1990), pp. 18–28.
[22] J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling, *A set of level* 3 *basic linear algebra subprograms*, ACM Trans. Math. Software, 16 (1990), pp. 1–17.
[23] J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson, *An extended set of Fortran basic linear algebra subroutines*, ACM Trans. Math. Software, 14 (1988), pp. 1–17.
[24] J. Dongarra, V. Eijkhout, and P. Łuszczek, *Recursive approach in sparse matrix LU factorization*, Sci. Programming, 9 (2001), pp. 51–60.
[25] J. J. Dongarra and D. W. Walker, *Software libraries for linear algebra computations on high performance computers*, SIAM Rev., 37 (1995), pp. 151–180.
[26] C. G. Douglas, M. Heroux, G. Slishman, and R. M. Smith, *GEMMW: A portable level* 3 *BLAS Winograd variant of Strassen's matrix multiply algorithm*, J. Comput. Phys., 110 (1994), pp. 1–10.
[27] E. Elmroth and F. G. Gustavson, *New serial and parallel recursive QR factorization algorithms for SMP systems*, in Applied Parallel Computing: Large Scale Scientific and Industrial Problems, B. Kågström et al., eds., Lecture Notes in Comput. Sci. 1541, Springer-Verlag, New York, 1998, pp. 120–128.
[28] E. Elmroth and F. G. Gustavson, *Applying recursion to serial and parallel QR factorization leads to better performance*, IBM J. Res. Develop., 44 (2000), pp. 605–624.
[29] E. Elmroth and F. G. Gustavson, *A faster and simpler recursive algorithm for the LAPACK routine DGELS*, BIT, 41 (2001), pp. 936–949.
[30] E. Elmroth and F. G. Gustavson, *High-performance library software for QR factorization*, in Applied Parallel Computing: New Paradigms for HPC in Industry and Academia, T. Sørvik et al., eds., Lecture Notes in Comput. Sci. 1947, Springer-Verlag, New York, 2001, pp. 53–63.
[31] G. E. Forsythe and C. B. Moler, *Computer Solution of Linear Algebraic Systems*, Prentice-Hall, Englewood Cliffs, NJ, 1967.
[32] J. D. Frens and D. S. Wise, *QR factorization with Morton-ordered quadtree matrices for memory re-use and parallelism*, in Proceedings of the 2003 ACM Symposium on Principles and Practice of Parallel Programming, ACM SIGPLAN Notices, 38 (10) (2003), pp. 144–154.
[33] M. Frigo, *A fast Fourier transform compiler*, in Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices, 34 (3) (1999), pp. 169–180.
[34] M. Frigo and S. G. Johnson, *FFTW: An adaptive software architecture for the FFT*, in Proceedings of the 1998 IEEE International Conference on Acoustics Speech and Signal Processing, Vol. 3, IEEE Press, 1998, pp. 1381–1384.
[35] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, *Cache-oblivious algorithms*, in Proceedings of the 40th Annual IEEE Symposium on Foundations of Computer Science, New York, 1999, IEEE Computer Society, Los Alamitos, CA, 1999.
[36] K. A Gallivan, R. J. Plemmons, and A. H. Sameh, *Parallel algorithms for dense linear algebra computations*, SIAM Rev., 32 (1990), pp. 54–135.
[37] J. D. Gardiner, A. L. Laub, J. A. Amato, and C. B. Moler, *Solution of the Sylvester matrix equation* $AXB^T + CXD^T = E$, ACM Trans. Math. Software, 18 (1992), pp. 223–231.
[38] G. Golub, S. Nash, and C. Van Loan, *A Hessenberg-Schur method for the matrix problem* $AX + XB = C$, IEEE Trans. Automat. Control, AC-24 (1979), pp. 909–913.
[39] G. Golub and C. Van Loan, *Matrix Computations*, 3rd ed., Johns Hopkins University Press, Baltimore, MD, 1996.
[40] K. Goto and R. van de Geijn, *On Reducing TLB Misses in Matrix Multiplication*, Technical Report TR-2002-55, FLAME Working Note 9, Department of Computer Sciences, University of Texas at Austin, 2002.
[41] J. Gunnels, F. G. Gustavson, G. Henry, and R. van de Geijn, *Formal linear algebra methods environment (FLAME)*, ACM Trans. Math. Software, 27 (2001), pp. 422–455.

[42] J. A. Gunnels, D. S. Katz, E. S. Quintana-Orti, and R. van de Geijn, *Fault-Tolerant High-Performance Matrix-Matrix Multiplication*, FLAME Technical Report TR-2000-34, Working Note 2, Department of Computing Sciences, University of Texas at Austin, 2000.
[43] F. G. Gustavson, *Recursion leads to automatic variable blocking for dense linear-algebra algorithms*, IBM J. Res. Develop., 41 (1997), pp. 737–755.
[44] F. G. Gustavson, *New generalized data structures for matrices lead to a variety of high performance algorithms*, in The Architectures for Scientific Software, R. F. Boisvert and P. T. P. Tang, eds., IFIP Conference Proceedings 188, Kluwer Academic, Dordrecht, The Netherlands, pp. 211–234.
[45] F. G. Gustavson, *High-performance linear algebra algorithms using new generalized data structures for matrices*, IBM J. Res. Develop., 47 (2003), pp. 31–554.
[46] F. G. Gustavson, A. Henriksson, I. Jonsson, B. Kågström, and P. Ling, *Recursive blocked data formats and BLAS's for dense linear algebra algorithms*, in Applied Parallel Computing: Large Scale Scientific and Industrial Problems, B. Kågström et al., eds., Lecture Notes in Comput. Sci. 1541, Springer-Verlag, New York, 1998, pp. 195–206.
[47] F. G. Gustavson, A. Henriksson, I. Jonsson, B. Kågström, and P. Ling, *Superscalar GEMM-based level* 3 *BLAS—The on-going evolution of a portable and high-performance library*, in Applied Parallel Computing: Large Scale Scientific and Industrial Problems, B. Kågström et al., eds., Lecture Notes in Comput. Sci. 1541, Springer-Verlag, New York, 1998, pp. 207–215.
[48] F. G. Gustavson and I. Jonsson, *Minimal-storage high-performance Cholesky factorization via blocking and recursion*, IBM J. Res. Develop., 44 (2000), pp. 823–849.
[49] W. W. Hager, *Condition estimates*, SIAM J. Sci. Statist. Comput., 5 (1984), pp. 311–316.
[50] S. J. Hammarling, *Numerical solution of the stable, non-negative definite Lyapunov equation*, IMA J. Numer. Anal., 2 (1982), pp. 303–323.
[51] A. Henriksson and I. Jonsson, *High-Performance Matrix Multiplication on the IBM SP High Node*, Master's thesis, UMNAD-98.235, Department of Computing Science, Umeå University, Umeå, Sweden, 1998.
[52] N. J. Higham, *Fortran codes for estimating the one-norm of a real or complex matrix with applications to condition estimation*, ACM Trans. Math. Software, 14 (1988), pp. 381–396.
[53] N. J. Higham, *Perturbation theory and backward error for* $AX - XB = C$, BIT, 33 (1993), pp. 124–136.
[54] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, SIAM, Philadelphia, 1996.
[55] IBM, *Engineering and Scientific Subroutine Library, Guide and Reference*, 1990.
[56] IBM, *Engineering and Scientific Subroutine Library, Guide and Reference*, Ver. 3, Rel. 1, 1998.
[57] IBM, *Engineering and Scientific Subroutine Library, Guide and Reference*, Ver. 3, Rel. 3, 2001.
[58] D. Irony, G. Shklarski, and S. Toledo, *Parallel and fully recursive multifrontal supernodal sparse Cholesky*, in Computational Science - ICCS 2002, P. Sloot et al., eds., Lecture Notes in Comput. Sci. 2330, Springer-Verlag, Berlin, 2002, pp. 335–344.
[59] I. Jonsson, *Analysis of Processor and Memory Utilization of Recursive Algorithms for Sylvester-Type Matrix Equations Using Performance Monitoring*, Technical Report UMINF-03.16, Department of Computing Science, Umeå University, Umeå, Sweden, 2003.
[60] I. Jonsson and B. Kågström, *Parallel triangular Sylvester-type matrix equation solvers for SMP systems using recursive blocking*, in Applied Parallel Computing: New Paradigms for HPC Industry and Academia, T. Sørvik et al., eds., Lecture Notes in Comput. Sci. 1947, Springer-Verlag, New York, 2001, pp. 64–73.
[61] I. Jonsson and B. Kågström, *Parallel two-sided Sylvester-type matrix equation solvers for SMP systems using recursive blocking*, in Applied Parallel Computing: Advanced Scientific Computing, J. Fagerhom et al., eds., Lecture Notes in Comput. Sci. 2367, Springer-Verlag, New York, 2002, pp. 297–306.
[62] I. Jonsson and B. Kågström, *Recursive blocked algorithms for solving triangular systems—Part* I: *One-sided and coupled Sylvester-type matrix equations*, ACM Trans. Math. Software, 28 (2002), pp. 392–415.
[63] I. Jonsson and B. Kågström, *Recursive blocked algorithms for solving triangular systems—Part* II: *Two-sided and generalized Sylvester and Lyapunov equations*, ACM Trans. Math. Software, 28 (2002), pp. 416–435.
[64] I. Jonsson and B. Kågström, *RECSY—A High Performance Library for Sylvester-Type Matrix Equations*, http://www.cs.umu.se/research/parallel/recsy, 2003.
[65] B. Kågström, *A perturbation analysis of the generalized Sylvester equation* $(AR - LB, DR - LE) = (C, F)$, SIAM J. Matrix Anal. Appl., 15 (1994), pp. 1045–1060.

[66] B. Kågström, P. Ling, and C. Van Loan, *GEMM-based level* 3 *BLAS: High-performance model implementations and performance evaluation benchmark*, ACM Trans. Math. Software, 24 (1998), pp. 268–302.

[67] B. Kågström, P. Ling, and C. Van Loan, *Algorithm* 784: *GEMM-based level* 3 *BLAS: Portability and optimization issues*, ACM Trans. Math. Software, 24 (1998), pp. 303–316.

[68] B. Kågström and P. Poromaa, *Distributed and shared memory block algorithms for the triangular Sylvester equation with* sep^{-1} *estimators*, SIAM J. Matrix Anal. Appl., 13 (1992), pp. 90–101.

[69] B. Kågström and P. Poromaa, *Computing eigenspaces with specified eigenvalues of a regular matrix pair* (A, B) *and condition estimation: Theory, algorithms and software*, Numer. Algorithms, 12 (1996), pp. 369–407.

[70] B. Kågström and P. Poromaa, *LAPACK–style algorithms and software for solving the generalized Sylvester equation and estimating the separation between regular matrix pairs*, ACM Trans. Math. Software, 22 (1996), pp. 78–103.

[71] B. Kågström and P. Van Dooren, *A generalized state-space approach for the additive decomposition of a transfer matrix*, Internat. J. Numer. Linear Algebra Appl., 1 (1992), pp. 165–181.

[72] B. Kågström and L. Westin, *Generalized Schur methods with condition estimators for solving the generalized Sylvester equation*, IEEE Trans. Automat. Control, 34 (1989), pp. 745–751.

[73] M. S. Lam, E. E. Rothberg, and M. E. Wolf, *The cache performance and optimizations of blocked algorithms*, in Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, 1991, pp. 63–74.

[74] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh, *Basic linear algebra subprograms for Fortran usage*, ACM Trans. Math. Software, 5 (1979), pp. 308–323.

[75] A. Lindkvist, *High-Performance Recursive BLAS Kernels Using New Data Formats for the QR Factorization*, Master's thesis, UMNAD-235.00, Department of Computing Science, Umeå University, Umeå, Sweden, 2000.

[76] MathWorks, *Using MATLAB*, The MathWorks Inc., Natick, MA, 2002.

[77] N. Park, B. Hong, and V. K. Prasanna, *Tiling, block data layout, and memory hierarchy performance*, IEEE Trans. Parallel Distrib. Systems, 14 (2003), pp. 640–654.

[78] B. N. Parlett and Y. Wang, *The influence of the compiler on the cost of mathematical software—In particular on the cost of triangular factorization*, ACM Trans. Math. Software, 1 (1975), pp. 35–46.

[79] T. Penzl, *Numerical solution of generalized Lyapunov equations*, Adv. Comput. Math., 8 (1998), pp. 33–48.

[80] P. Poromaa, *Parallel algorithms for triangular sylvester equations: Design, scheduling and scalability issues*, in Applied Parallel Computing: Large Scale Scientific and Industrial Problems, B. Kågström et al., eds., Lecture Notes in Comput. Sci. 1541, Springer-Verlag, New York, 1998, pp. 438–446.

[81] P. Poromaa, *High Performance Computing: Algorithms and Library Software for Sylvester Equations and Certain Eigenvalue Problems with Applications in Condition Estimation*, Ph.D. Thesis, UMINF-97.16, Department of Computing Science, Umeå University, Umeå, Sweden, 1997.

[82] E. Rabani and S. Toledo, *Out-of-core SVD and QR decompositions*, in Proceedings of the 10th SIAM Conference on Parallel Processing for Scientific Computing, Portsmouth, VA, CD-ROM, SIAM, Philadelphia, 2001.

[83] E. Rabani and S. Toledo, *Very large electronic structure calculations using an out-of-core filter-diagonalization method*, J. Comput. Phys., 180 (2002), pp. 256–269.

[84] H. Sagan, *Space-Filling Curves*, Springer-Verlag, Berlin, 1994.

[85] J. K. Salmon and M. S. Warren, *Skeletons from the treecode closet*, J. Comput. Phys., 111 (1994), pp. 136–155.

[86] J. K. Salmon, M. S. Warren, and G. S. Winckelmans, *Fast parallel tree codes for gravitational and fluid dynamical n-body problems*, Internat. J. Supercomput. Appl., 8 (1994), pp. 129–142.

[87] H. Samet, *The quadtree and related hierarchical data structures*, Comput. Surveys, 16 (1984), pp. 188–260.

[88] R. Schreiber and C. Van Loan, *A storage-efficient WY representation for products of Householder transformations*, SIAM J. Sci. Statist. Comput., 10 (1989), pp. 53–57.

[89] SGI, *Scientific Computing Software Library (SCSL)*, software and documentation available from http://www.sgi.com/software/scsl.html, 1993–2003.

[90] SLICOT, *The SLICOT Library and the Numerics in Control Network (NICONET) website*, http://www.win.tue.nl/niconet/.

[91] B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema, and C. B. Moler, *Matrix Eigensystem Routines–EISPACK Guide*, Lecture Notes in Comput. Sci. 6, Springer-Verlag, Berlin, 1976.

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

[93] G. W. Stewart and J.-G. Sun, *Matrix Perturbation Theory*, Academic Press, New York, 1990.

[94] V. Strassen, *Gaussian elimination is not optimal*, Numer. Math., 13 (1969), pp. 354–356.

[95] S. Toledo, *Locality of reference in LU decomposition with partial pivoting*, SIAM J. Matrix Anal. Appl., 18 (1997), pp. 1065–1081.

[96] TOP500, *The Top* 500 *Supercomputer Sites*, http://www.top500.org/.

[97] V. Valsalam and A. Skjellum, *A framework for high-performance matrix multiplication based on hierarchical abstractions, algorithms and optimized low-level kernels*, Concurrency Computat. Pract. Exper., 14 (2002), pp. 805–839.

[98] R. Vuduc, J. W. Demmel, and J. A. Bilmes, *Statistical models for automatic performance tuning*, in Proceedings of the International Conference on Computational Science, Lecture Notes in Comput. Sci. 2073, Springer-Verlag, New York, 2001, pp. 117–126.

[99] R. Vuduc, A. Gyulassy, J. W. Demmel, and K. A. Yelick, *Memory hierarchy optimization and bounds for sparse* $A^T Ax$, in Proceedings of the ICCS Workshop on Parallel Linear Algebra, Lecture Notes in Comput. Sci. 2660, Springer-Verlag, New York, 2003, pp. 705–714.

[100] R. C. Whaley, A. Petitet, and J. Dongarra, *Automated empirical optimization of software and the ATLAS project*, LAPACK Working Note 147, 2000; see also http://sourceforge.net/projects/math-atlas/.

[101] J. H. Wilkinson and C. Reinsch, *Handbook for Automatic Computation, Vol.* II. *Linear Algebra*, Springer-Verlag, New York, 1971.

[102] D. S. Wise, G. A. Alexander, J. D. Frens, and Y. H. Gu, *Language support for Morton order matrices*, ACM SIGPLAN Notices, 36 (7) (2001), pp. 24–33.

[103] M. E. Wolf and M. S. Lam, *A data locality optimizing algorithm*, in Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation, 1991, pp. 30–44.

[104] Q. Yi, V. Adve, and K. Kennedy, *Transforming loops to recursion for multi-level memory hierarchies*, ACM SIGPLAN Notices, 35 (5) (2000), pp. 169–181.