Recursive Blocked Algorithms, Data Structures, and High-Performance Software for Solving Linear Systems and Matrix Equations

Isak Jonsson

Ph.D. Thesis, 2003
UMINF-03.17

Department of Computing Science
Umeå University
SE-901 87 Umeå, Sweden

Akademisk avhandling som med tillstånd av rektorsämbetet vid Umeå universitet framläggs till offentlig granskning onsdagen den 17 december klockan 13.15 i sal MA121, MIT-huset, för avläggande av teknologie doktorsexamen.

Print & Media, Umeå Universitet
UMINF-03.17 ISSN-0348-0542 ISBN 91-7305-568-9

Abstract

This thesis deals with the development of efficient and reliable algorithms and library software for factorizing matrices and solving matrix equations on high-performance computer systems. The architectures of today's computers consist of multiple processors, each with multiple functional units. The memory systems are hierarchical with several levels, each having different speed and size. The practical peak performance of a system is reached only by considering all of these characteristics. One portable method for achieving good system utilization is to express a linear algebra problem in terms of level 3 BLAS (Basic Linear Algebra Subprogram) transformations. The most important operation is GEMM (GEneral Matrix Multiply), which typically defines the practical peak performance of a computer system. There are efficient GEMM implementations available for almost any platform, thus an algorithm using this operation is highly portable.

The dissertation focuses on how recursion can be applied to solve linear algebra problems. Recursive linear algebra algorithms have the potential to automatically match the size of subproblems to the different memory hierarchies, leading to much better utilization of the memory system. Furthermore, recursive algorithms expose level 3 BLAS operations, and reveal task parallelism. The first paper handles the Cholesky factorization for matrices stored in packed format. Our algorithm uses a recursive packed matrix data layout that enables the use of high-performance matrix–matrix multiplication, in contrast to the standard packed format. The resulting library routine requires half the memory of full storage, yet the performance is better than for full storage routines.

Paper two and tree introduce recursive blocked algorithms for solving triangular Sylvester-type matrix equations. For these problems, recursion together with superscalar kernels produce new algorithms that give 10-fold speedups compared to existing routines in the SLICOT and LAPACK libraries. We show that our recursive algorithms also have a significant impact on the execution time of solving unreduced problems and when used in condition estimation. By recursively splitting several problem dimensions simultaneously, parallel algorithms for shared memory systems are obtained. The fourth paper introduces a library—RECSY—consisting of a set of routines implemented in Fortran 90 using the ideas presented in paper two and three. Using performance monitoring tools, the last paper evaluates the possible gain in using different matrix blocking layouts and the impact of superscalar kernels in the RECSY library.

Preface

The thesis consists of the following five papers and an introduction including a summary of the papers.

I. Fred G. Gustavson and Isak Jonsson. Minimal-storage high-performance Cholesky factorization via blocking and recursion*. *IBM Journal of Research and Development*, Vol. 44:6, 2000.

II. Isak Jonsson and Bo Kågström. Recursive Blocked Algorithms for Solving Triangular Systems–Part I: One-Sided and Coupled Sylvester-Type Matrix Equations†. *ACM Transactions on Mathematical Software*, Vol 28:4, 2002.

III. Isak Jonsson and Bo Kågström. Recursive Blocked Algorithms for Solving Triangular Systems–Part II: Two-Sided and Generalized Sylvester and Lyapunov Matrix Equations†. *ACM Transactions on Mathematical Software*, Vol 28:4, 2002.

IV. Isak Jonsson and Bo Kågström. RECSY — A High Performance Library for Sylvester-Type Matrix Equations‡. In Kosch et al. (eds), *Euro-Par 2003, Parallel Processing*, Lecture Notes in Computer Science, Vol 2790, Springer-Verlag, 2003.

V. Isak Jonsson. Analysis of Processor and Memory Utilization of Recursive Algorithms for Sylvester-Type Matrix Equations using Performance Monitoring. Report UMINF-03.16, Department of Computing Science, Umeå University, 2003.

In Paper I, a recursive algorithm for the Cholesky factorization is introduced. The algorithm uses a recursive packed data format optimized for this algorithm, together with optimized kernels that provide good performance also for small-sized problems.

The topic of papers II–V is the efficient solution of Sylvester-type matrix equations. Paper II and Paper III introduce novel recursive-blocked algorithms

*Reprinted by permission of the IBM Journal of Research and Development, White Plains, New York. © 2000 IBM.

†Reprinted by permission of ACM, New York. © 2002 The Association for Computing Machinery.

‡Reprinted by permission of Springer-Verlag, Berlin. © 2003 Springer-Verlag.

for one-sided and two-sided Sylvester-type matrix equations, respectively. Paper IV presents a software library based on the algorithms in papers II–III. Finally, Paper V provides an in-depth analysis of different performance issues relating to recursive blocked algorithms using performance monitoring tools.

Populärvetenskaplig sammanfattning på svenska

Snabbare rutiner för att lösa stora matematiska ekvationssystem och matrisekvationer

Syftet med den forskning som ligger bakom denna avhandling, dvs analys och utveckling av algoritmer för att lösa ekvationssytem och matrisekvationer, är att bättre utnyttja datorresurser för att kunna lösa stora och komplicerade problem.

De problem som vi försöker lösa hittar man ofta inom reglerteori och reglerteknik. Givet en matematisk modell för en process vill man undersöka vilka egenskaper modellen och den bakomliggande processen har. Ett exempel på en komplicerad process är en pappersmassaindustri eller ett stort elnät. De stora strömavbrotten i USA de senaste åren är ett typiskt fall på mycket stora system (processer) som är svåra att styra. På samma sätt förklarades JAS-kraschen på Långholmen 1993 med att styrsystemet var för instabilt. De rutiner vi utvecklar kan bland annat användas till att analysera sådana system.

Standardmetoderna som idag används för att lösa problemen ovan är ofta alldeles för långsamma. Anledningen till detta återfinns i dagens datorers uppbyggnad. Man kan likna en dators minne vid närminnet och långtidsminnet hos oss själva. Medan närminnet kan ta fram information på ett ögonblick, rymmer det inte så mycket. Långtidsminnet däremot rymmer information från hela livet, men det tar lite längre tid att få fram. På samma sätt fungerar datorer. De snabba registren och cache-minnena fungerar som närminne, och de långsammare primärminnena och sekundärminnena fungerar som långtidsminne. Standardalgoritmerna utnyttjar registren och cache-minnena dåligt och måste hela tiden fråga primärminnet eller sekundärminnet efter information, vilket ger en icke önskad overhead. För att få bra prestanda gäller det att utveckla metoder och programvara som hanterar och använder minneshierarkin i dagens datorsystem på bästa möjliga sätt.

Lösningen hittar man genom att dela in problemet i flera mindre delar, blockning, där varje liten del kan lösas för sig. Data för varje delproblem hämtas från primärminnet till cache-minnena, och man löser ett mindre delproblem i taget. När man är klar med ett delproblem hämtas data till nästa delproblem.

En av svårigheterna är att det finns många nivåer av minnen som problemet måste blockas för, vilket leder till så kallad hierarkisk blockning.

De artiklar som ingår i den här avhandlingen beskriver hur man kan använda rekursion för att automatiskt hitta rätt storlek på delproblemen. Man löser uppgiften genom att dela problemet i två delproblem som är ungefär lika stora. Dessa problem delas ytterligare en gång, och så vidare tills dess att delproblemen når en minsta storlek. Genom denna delning hittar man automatiskt en storlek som passar det minsta cache-minnet (level 1 cache), en annan storlek som passar nästa nivå i datorns minneshierarki (level 2 cache) och så vidare.

De nya algoritmer och programvara som presenteras i avhandlingen är ofta flera gånger snabbare än de metoder och rutiner som används idag. Detta innebär att man kan använda dem för att lösa större problem svarande mot mer realistiska matematiska modeller, vilket i sin tur kan leda till bättre styrning av komplicerade processer såsom elnät.

Acknowledgements

First of all, I would like to thank Professor Bo Kågström, who is the co-author of three of the papers and also my research and thesis advisor. He has been very important in showing me what trees that might bear fruit. Trying to convince him of the pros or cons of an algorithm has forced me to think of all aspects of the specific problem. In short, he has given me the inspiration needed, the necessary pressure, and first-rate conditions for writing this thesis.

I also send a big thanks to Professor Fred Gustavson, who is the co-author of the first paper in this thesis. He has taught me the value of never neglecting the small details of an algorithm. His ever-lasting urn of brain-teasing problems has helped me to keep the spirit, although I tend to cheat and use brute force to solve the puzzles.

Special thanks to my family. I am fortunate to have my wife Maria, who understands the endurance it means to be a PhD student and to write theses. We have tried to help each other, and I think we have learned quite a lot. Thanks go to my parents, my sister, my brothers, and the rest of my family who have helped Prof. Kågström by asking that dreadful question: “When will you be finished?” Pressure is something all PhD students need, but no one would admit it. My appreciation extends to my friends as well.

The Department of Computing Science at Umeå University is a veritable melting pot of ideas. In the same way as the brain-teasing problems, the discussions at the lunch table demand you to sharpen your mind, and to be open for new ideas.

The research groups in Parallel and High-Performance Computing and in Numerical Linear Algebra are very vigorous and I am lucky to be a member. I would also like to show my gratefulness to the staff at HPC2N, who have provided me with state-of-the-art HPC systems. Most results in the thesis are concerned with algorithms and optimization techniques, which could not have been developed and evaluated without a working computer environment.

Umeå, November 2003

Isak Jonsson

Contents

1 Introduction to high-performance computing and software **1**

1.1 High performance linear algebra software libraries 2

1.2 The equations 3

1.3 Algorithm techniques 3

1.3.1 Recursive blocked algorithms 4

1.3.2 Recursive blocked data formats 5

1.3.3 The base case: superscalar kernels 6

2 Contributions **9**

2.1 Paper I 9

2.2 Paper II 10

2.3 Paper III 11

2.4 Paper IV 12

2.5 Paper V 13

References **15**

Paper I **23**

Paper II **55**

Paper III **83**

Paper IV **107**

Paper V **121**

Chapter 1

Introduction to high-performance computing and software

Already in 1965, Intel co-founder Gordon Moore [15] presented the law that would permeate the conditions for developments in high-performance computing: the number of transistors that fit on a die will grow exponentially with time. The coefficients of the law were later adjusted to a doubling every second year. There are several interpretations; one is that the speed of a microprocessor also grows exponentially. This interpretation should not be taken for granted, though. While it is relatively easy to reach the maximum performance of an integrated circuit from 1965 – much easier than actually finding a working I.C. from 1965, it is harder to get close to the maximum performance of a super-scalar, super-pipelined, multi-core processor chip of today, almost 40 years later. Part of the explanation to this phenomenon can be found in Moore's article, as he sees into the future: "For example, memories built of integrated electronics may be distributed throughout the machine instead of being concentrated in a central unit".

Memories in modern computer architectures are organized in a hierarchy. At the top of the hierarchy are the registers, with virtually zero access time. The registers, however, usually only holds 8–128 numbers. Below the registers are cache memories: small but fast memories. There are typically several levels of cache memories, where the smallest (level 1 cache) – but fastest one – is closest to the processing unit. Below the cache memories is the primary memory. In a multi-processor computer, the primary memory may either be shared by the processors, or each processor has a local primary memory (called distributed memory). Below the primary memory are the secondary memories (hard disks, tape) and remote memories. The reason for this large hierarchy is two-folded. One cause is the cost for developing different memory types. Fast memories are expensive, so they tend to be small. Another cause is technology issues. In order for a memory to be fast, it has to be very close to the processor. In fact, the first levels of cache memories are usually on the same die as the processing unit. Obviously, there is a physical limit on the size of these cache memories.

With this background, the motivation for the work in this thesis is almost completely defined. Given this complex hardware organization, how to solve large matrix equations fast and with sustained precision and stability?

1.1 High performance linear algebra software libraries

The ever-growing demand of solving larger problems puts requirements on the software used. In order to make the software run near the peak performance of the target architecture, it is necessary to optimize the code (i.e., tune) for the machine. However, this task is both tedious and requires detailed knowledge about the memory architecture, the register set, and the behavior of the functional units of the processor.

Linear algebra operations and equations are distinctive building blocks that can be implemented or solved using a series of matrix transformation primitives. This is where BLAS, or the Basic Linear Algebra Subprograms, appears on the scene. BLAS is a set of matrix transformation routines, available for Fortran 77[*] programs. By using BLAS, the complexity of the architecture is encapsulated in simple building blocks.

The first set of BLAS routines, level 1 BLAS [38] includes vector–scalar and vector–vector operations. This was the logical choice of building blocks given the fact that most computers at the time of introduction were vector computers with specialized vector instructions. However, the level 1 BLAS operations only execute a small number of operations every access to memory, and thus do not come close to peak performance on machines with deep memory hierarchies. With the introduction of level 2 BLAS [9] for matrix–vector operations and later level 3 BLAS [10, 35] for matrix–matrix operations, the building blocks approached the theoretical peak of the computer. The level 3 BLAS routines are dissimilar to the other sets, as level 3 BLAS routines, e.g., matrix multiplications, involves $O(n)$ times more operations than data elements. For the other two sets, the complexity of the operations and the number of data elements are of the same order.

Given a high-performance implementation of BLAS, the task is then to utilize level 3 BLAS operations in a great extent. This is done for important linear algebra operations in the standard libraries (LAPACK [2], SLICOT [41],

[*]Most BLAS libraries are also available for C, C++, and Fortran 90 programs.

ESSL), but not all. The algorithms and implementations presented in this thesis fill gaps in these libraries, and show that recursion is a means to create software libraries which are efficient, portable, and easy to maintain.

By making an effort to create algorithms that perform close to the practical peak, the expectation is that more users will find the routines worthwhile to use in their applications.

1.2 The equations

In Table 1.1, the factorization and equations considered in this thesis is listed. The Cholesky factorization is a central tool in linear algebra, and is a key element in matrix inversion, symmetric definite generalized eigenvalue problems etc. Among the applications are finite-element methods. The Sylvester-type matrix equations appear in different control theory applications, e.g., stability problems, model reduction, balancing, H_∞ control. The first three matrix equations in Table 1.1 are called one-sided and the five last matrix equations are called two-sided (see Chapter 2 for the motivation for this classification). The acronyms CT and DT are used for continuous-time and discrete-time systems, respectively.

Name	Matrix equation/factorization	Paper
Cholesky	$A = LL^T$ $A = A^T$ positive definite	I
Standard Sylvester (CT)	$AX - XB = C$	II
Standard Lyapunov (CT)	$AX + XA^T = C$	II
Generalized coupled Sylvester	$(AX - YB, DX - YE) = (C, F)$	II
Standard Sylvester (DT)	$AXB^T - X = C$	III
Standard Lyapunov (DT)	$AXA^T - X = C$	III
Generalized Sylvester	$AXB^T - CXD^T = E$	III
Generalized Lyapunov (CT)	$AXE^T + EXA^T = C$	III
Generalized Lyapunov (DT)	$AXA^T - EXE^T = C$	III

Table 1.1: Cholesky factorization, one-sided (top) and two-sided (bottom) Sylvester-type matrix equations.

1.3 Algorithm techniques

In this section, we describe the different techniques used to reach close to peak performance.

1.3.1 Recursive blocked algorithms

In the first section of the chapter, the memory hierarchy was described. From the characteristics of the memory hierarchy follows the importance of a good memory reference pattern. If an algorithm keeps the memory accesses confined to a small number of elements at the time, the algorithm obtains good performance, as most accesses will be carried out by the cache memories. On the other hand, if the memory accesses are disperse, a large part of the accesses will reach the primary memory, and the algorithm will suffer from access latencies, called cache miss penalties.

In order to manage a complex memory system we apply hierarchical blocking. The traditional technique is called explicit multi-level blocking. Typically, loop nests are split into several nests explicitly, each loop nest matching one level of the memory hierarchy. While it is possible to match every level of the memory hierarchy in this way, it requires deep knowledge of the host architecture. For example, a matrix multiplication routine, which is explicitly blocked to match three levels of cache memories requires twelve nested do-loops. Each of these loops needs to be tuned for the specific architecture.

Instead of using explicit blocking to achieve good memory access patterns, recursive blocking is used in the papers in this thesis. Recursion is an old yet powerful technique which uses self-similarities to divide the problem into smaller parts. In order to reformulate a problem using recursion, the following must be defined:

- How to split the problem; i.e., where is the natural splitting point (divide).
- How to calculate the answer to the large problem given the answers to the smaller problems (conquer).
- What is the base case, and how to solve it (use kernels).

We call a set of answers to these problems a recursive template. As an example of a recursive template, consider again matrix multiplication $C = A \cdot B$, where A, B are dense matrices. Now, define the recursion as follows. Split A and B in the middle into submatrices $[A_1, A_2]$ and $[B_1; B_2]$ (Matlab notation), where A_1 and A_2 differs at most by one column. The same goes for the rows of B_1 and B_2. Now, recursion can be applied to calculate $A_1 \cdot B_1$ and $A_2 \cdot B_2$. The base case is reached when A is a single column (and B is a single row). The answer to the base case is the outer product of column A and row B. The answer to the large problem is to add $A_1 \cdot B_1$ and $A_2 \cdot B_2$. This example, used for illustration only, will not give a good memory access pattern, as only one out of three problem dimensions is split. In the recursive templates given in the papers, we always divide the largest dimension, or several at the same time. By doing this, smaller problems are kept squarish, and good temporal locality is obtained.

In practice, continuing recursion down to a single element or column damages performance. For small problems, the overhead of the recursive function calls becomes considerable compared to the actual operations. Also, when the problem fits in the level 1 cache, the temporal locality is irrelevant. For that reason, the base case is defined as when the problem fits into level 1 cache. The problem is then solved using a superscalar kernel, see Section 1.3.3.

For the matrix equations solved in this thesis, there is also another benefit from the recursive formulation. It effectively reveals matrix-matrix level 3 BLAS operations [10]. Thus, most of the work in the recursive algorithm will be performed by efficient BLAS library routines, optimized for the user's host architecture.

Historically, the use of recursion has led to several algorithms with lower complexity than the standard algorithms. The most famous are the Fast Fourier Transform (FFT) by Cooley and Tukey [8] and Strassen's algorithm for matrix multiplication [42]. Both these algorithms have lower complexity than the standard algorithms for these computational tasks. On the other hand, blocked linear algebra routines sometimes lead to higher complexity (larger coefficients of the leading terms) due to computation of larger temporaries, typically level 3 operations. Examples of this are shown in Paper III. Note that this is not due to recursion itself, but has to do with the blocking of the algorithm. By choosing the splitting point, this amount of overhead can be adjusted, on the expense of the blocking features. For an example of such a modification of a recursive algorithm for the QR factorization, see Elmroth and Gustavson [11]. In [29], Jonsson and Kågström show that the recursive blocked one-sided Sylvester solvers have lower complexity than the standard methods.

1.3.2 Recursive blocked data formats

The recursive blocked algorithms mainly improve on the temporal locality, which means that blocks (submatrices) that recently have been accessed will most likely be referenced soon again. 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) nearby 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 avoid unnecessary data transfers in the memory hierarchy.

In general, and similar to the splittings of a recursive algorithm, the recursive data format should not be repeated down to a single element. This is because calculating the address of an element is more expensive with a recursive data format. Instead, the matrices should be divided into fixed size submatrices, where the elements in the submatrices are stored in standard order. The mutual order of these submatrices is then resolved using recursion. For an example of a recursive data format, see Figure 1.1.

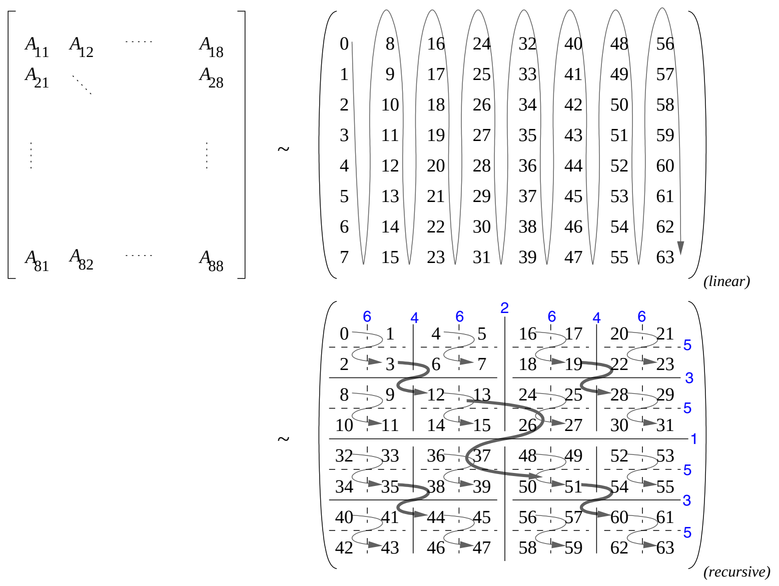

Figure 1.1: The mapping of an 8×8 block matrix in linear block column order and recursive block row order. The three levels of recursive splitting are labeled 1-2, 3-4 and 5-6.

The recursive data format has two penalties that must be considered. The first is the overhead of data transformations. In general, this results in longer code, some extra execution time and memory requirements for the transformation between input/output data and the internal blocked data format. However, for large-scale matrix computations this issue is only marginal. The second is the limitation of using standard linear algebra operations (BLAS), such as matrix multiplication. From the recursive templates above, we obtain algorithms that do most of its work in large, efficient calls to general matrix multiply and add (GEMM) operations. With the recursive blocked data format, there are no large submatrices that can be passed to a GEMM routine. Instead, there are several calls to BLAS routines, each with small-sized data blocks chosen to fit the level 1 cache.

1.3.3 The base case: superscalar kernels

Recursion is one of several ways of handling blocking and obtaining good cache memory reuse on a high level but may make a poor job at the bottom of the recursion tree where matrices are small or even single elements. There are a number of reasons for this. One is the overhead for the actual recursive call. Setting up the stack and passing arguments takes much more processor cycles than the actual floating point operation. Another cause is that most compilers make a poor job of generating assembler instructions that match a superscalar architecture with several registers and a long instruction pipeline. Because of the performance impact of pursuing the recursion to a single element, the algorithms in this thesis end the recursion when the acquired subproblem is smaller than a certain size.

In Paper I, the base case is reached and recursion is ended when the problem size is less than 32. Problems smaller than this size are solved using a superscalar kernel that implements an iterative algorithm with outer loop unrolling which enables a high utilization of all different floating point units in the processor.

Paper II and Paper III also introduce specialized kernels for the respective problems. Here, we introduce fast matrix multiplication kernels, which with low overhead handles small matrix multiplications. This enables us to do recursion further, down to blocks of size 4×4, when the Sylvester-type matrix equation kernels solve the small-sized equation.

Chapter 2

Contributions

This chapter gives a brief summary of the papers in this thesis. Three of the papers (I–III) are published in well-known scientific journals, one is a refereed conference proceeding paper (IV), and the last contribution is a technical report (V).

Moreover, results in the thesis and earlier results related to the topic of the thesis have been presented at several international conferences, e.g., see [18, 19, 20, 21, 26, 29].

2.1 Paper I

In this contribution, which continues work from Anderson et al. [3], a recursive algorithm for the Cholesky factorization is presented. The algorithm computes $A = LL^T$ for a symmetric positive definite A, where A is stored in packed format. The factorization is used for solving symmetric positive systems of equations $Ax = b$, or for computing the inverse of A. While the packed format saves half of the memory compared to full storage, it precludes the use of standard level 3 BLAS operations. Instead, packed format algorithms use level 1 and level 2 BLAS operations, and hence perform much worse than full storage algorithms.

The recursive algorithm in paper I uses a recursive format that requires the same memory footprint as the packed format. The format is constructed by dividing the triangle into three parts: two smaller triangles and one rectangle. The triangles themselves are divided down to single elements, whereas the rectangle is stored in standard format. Hence, operations on the rectangle can use level 3 routines. The paper describes the recursive Cholesky algorithm and how it relates to the recursive data format. Also, recursive symmetric rank-k and recursive triangular matrix solve routines for the recursive data format are presented, as they are level 3 building blocks for the factorization. The paper gives details on how to transform A from packed to recursive data format, using a small temporary buffer. The data transformation also makes use of the symmetry to change the orientation of the equation. Then, the fastest variant of the matrix multiply operation, where elements are accessed with stride 1, can be used throughout the algorithm.

Another contribution in paper I is the superscalar kernels for factorization, rank-k update and triangular matrix solve. By terminating recursion prematurely, it is shown analytically that the overhead from recursion is negligible. However, as the format is recursively constructed all the way to element level, the superscalar kernels need to work with non-linear addressing. In the paper, three different methods that handle this addressing are presented, and their respective advantages are discussed. The most ingenious solution is look-up maps, which are calculated before the recursion takes place and stored very efficiently.

The routines developed in this contribution later became part of the IBM ESSL library.

2.2 Paper II

Paper II presents recursive blocked algorithms for solving one-sided Sylvester-type matrix equations. These equations include the continuous-time Sylvester equation $AX - XB = C$, the continuous-time Lyapunov equation $AX + XB^T = C$, and the generalized coupled Sylvester matrix equation $(AX - YB, DX - YE) = (C, F)$. 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. The solvers are of Bartels–Stewart-type [4] and contains three main steps: (i) reduce the problem to quasi-triangular form; (ii) solve the quasi-triangular problem; (iii) retransform the solution to the original problem. The novel algorithms presented all concern the second step.

In the paper, we give an analysis of the impact of the performance of the kernel operations by modeling the amount of work of level 2 and level 3 operations. The analysis shows that poor performance of these kernels spoils the overall performance even for large problems. At first, this looks somewhat unexpected, as the kernels only count for $O(n^2)$ operations out of the total $O(n^3)$ work, so the weak performance of the kernels should not be noticeable. However, since the kernels perform more than 100 times slower than the matrix multiply, it has an impact even for large problems. In order to minimize the negative impact, we have developed new kernels that are more suitable for superscalar processors. These new kernels do not include scaling and complete pivoting. However, if a near-singularity is detected, the algorithm backtracks and uses another more robust but slower kernel which includes both scaling to avoid overflow and complete pivoting. In this way, the stability of the algorithm is maintained as well as the accuracy of the results.

As mentioned earlier, the recursive method reveals large level 3 operations, which are easily parallelized on a shared memory machine using an SMP BLAS library. Moreover, by splitting several dimensions of the problem simultaneously, independent tasks suitable for shared-memory parallelization become apparent. In the paper, we show how the algorithms can be parallelized using OpenMP. Although the triangular solve is only one step in the solution process, performance results in Paper II show that the overall performance of solving unreduced matrix equations is also improved a lot, and especially in connection with condition estimation.

Uniprocessor and SMP parallel performance results of our recursive blocked algorithms and corresponding routines in the state-of-the-art libraries LAPACK and SLICOT are presented. The performance improvements of our recursive algorithms are remarkable, including 10-fold speedups compared to standard algorithms.

We remark that the recursive blocked algorithms allow sliding splittings, with the splitting points varying between the second and the penultimate row and/or column. By not splitting in the middle, the algorithms exhibit different memory access patterns and non-square updates, which in general degrade the performance. In the extreme cases we obtain the standard algorithms.

A more detailed version of this paper is available as [27].

2.3 Paper III

In this paper, the work from Paper II is extended and generalized to solve two-sided matrix equations. These equations include the discrete-time Sylvester equation $AXB^T - X = C$, the discrete-time Lyapunov equation $AXA^T - X = C$, and generalized Sylvester and Lyapunov equations. Examples of the latter include the continuous-time and discrete-time generalized Lyapunov equations $AXE^T + EXA^T = C$ and $AXA^T + EXE^T = C$, respectively.

All these equations are called two-sided, since they include matrix product terms of type $\text{op}(A)X\text{op}(B)$, where as before $\text{op}(A)$ can be A or A^T. Some of these matrix equations can be seen as special cases of other formulations. In practice, all matrix equations are treated 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. The recursive blocked algorithms for two-sided equations require both extra workspace and extra computational work, which we briefly discuss below.

The computation of matrix triple product updates $\text{op}(A)X\text{op}(B)$ can be done in two ways, namely $(\text{op}(A)X)\text{op}(B)$ or $\text{op}(A)(X\text{op}(B))$. These two matrix products may need different number of floating point operations and we choose the order which minimizes the computational work. We also remark that the computation of a matrix triple product requires temporary workspace for storing intermediate matrix products of two (sub)matrices. In the recursive algorithms, the transformed A and B may turn out to be triangular, quasi-triangular, rectangular, trapezoidal, or quasi-trapezoidal. If $X = X^T$, even more clever algorithms are used which utilize the symmetry structure.

As a result, the recursive blocked algorithms for the two-sided matrix equations require more floating point operations than the standard column-wise algorithms, e.g., up to 56% more operations for the two-sided Lyapunov equations. Despite this overhead, the recursive blocked algorithms perform much faster in practice, which is an effect of their better data locality and higher level 3 BLAS ratio, and the use of new superscalar kernels.

Also in paper III, parallel algorithms are presented using OpenMP and parallel BLAS. A more detailed version of this paper is available as [28].

2.4 Paper IV

In paper IV, the software library RECSY is presented. RECSY contains serial and parallel implementations of the algorithms for solving one-sided and two-sided Sylvester-type matrix equations described in Paper II and Paper III. The library is written in Fortran 90. The use of Fortran 90 has the effect that constructs such as recursion and dynamic memory allocation can be implemented using basic language constructs, which simplify the code and make the library more user-friendly. In total, 42 different cases of eight equations (three one-sided and five three-sided) are solved by the library, either in serial or in parallel using OpenMP. The library includes superscalar kernels, much faster than traditional SLICOT or LAPACK kernels. However, the new kernels do no overflow and near-singularity checking. If the problem is ill-conditioned, the routine as an alternative backtracks and falls back to kernels that carry out complete pivoting. RECSY also includes a superscalar matrix multiplication routine with low overhead and setup cost, designed for the small- matrix multiplications done in the lower part (branches) of the recursion tree.

The RECSY routines including all of the kernels is written entirely in Fortran 90, so it is portable on a wide range of platforms, including Unix and Windows. In order to make the library easy to use, wrapper routines for SLICOT and LAPACK are included, so the user can keep his/her original code and simply link with RECSY. This means that the user calls the SLICOT routine for solving an unreduced problem, and the transformed quasi-triangular matrix equation is automatically solved by RECSY.

The RECSY library together with building instructions can be downloaded from `http://www.cs.umu.se/~isak/recsy`. For an overview of the routines in RECSY and its relation to routines in other libraries see Figure 2.1.

2.5 Paper V

Even though the work with RECSY has been rewarding with very good performance results, there is always an urge to save a few more cycles of the running time. As more detailed information was needed for proceeding with the development of recursive algorithms, a more thorough study of the behavior of the software on a given target architecture was needed. In the final paper of this thesis, performance monitoring tools are used to examine the properties of the recursive blocked algorithms in the RECSY library. While standard timing routines fail to give high resolution, performance monitoring tools use hardware registers to give low-latency results down to nanosecond precision. The tools can also give information not available by other methods, such as cache miss counts.

The paper looks into the aspects of data storage formats, and comparisons between using recursive blocked, linear blocked, and standard Fortran data formats are presented, including the cost of converting the input matrices to the blocked data formats. One result of the analysis shows the necessity of the superscalar matrix multiplication routine in the RECSY library.

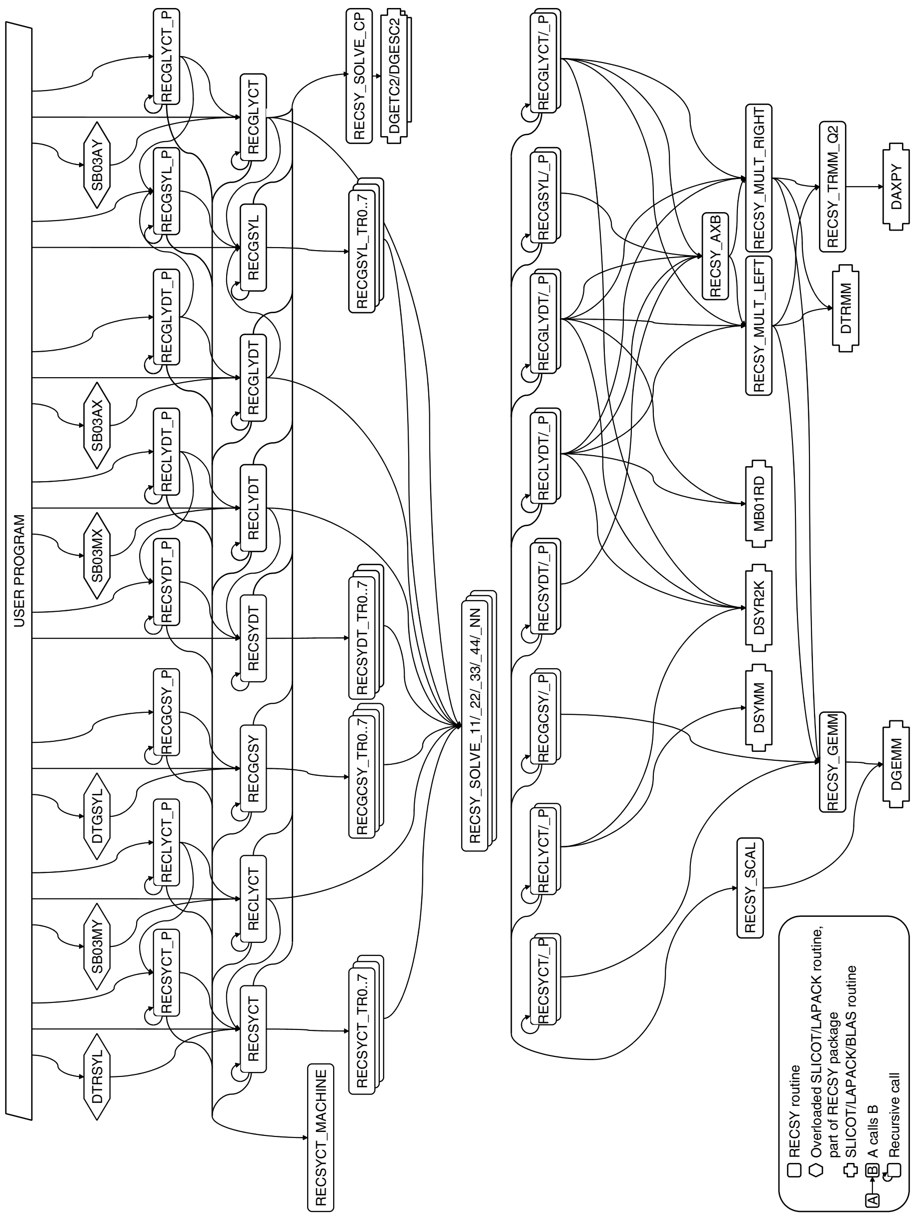

Figure 2.1: Call graph of the RECSY library. Overloaded SLICOT and LAPACK routines are shown as hexagons. Level 3 BLAS functions and auxiliary BLAS, LAPACK, and SLICOT routines used by RECSY are shown as crosses.

References

[1] N. Ahmed and K. Pingali. Automatic generation of block-recursive codes. In A. Bode et al., editors, *Euro-Par 2000 Parallel Processing*, volume 1900 of *Lecture Notes in Computer Science*, pages 368–378. Springer-Verlag, 2000.

[2] 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 User's Guide.* SIAM, Philadelphia, third edition, 1999.

[3] B. Andersen, F. Gustavson, and J. Waśniewski. A recursive formulation of Cholesky factorization of a matrix in packed storage. *ACM Trans. Math. Softw.*, 27(2):214–244, June 2001.

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

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

[6] S. Chatterjee, A. R. Lebeck, P. K. Patnala, and M. Thottethodi. Recursive Array Layouts and Fast Matrix Multiplication, *IEEE Trans. Parallel and Distributed Systems.* 13(11):1105–1123.

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

[8] J.W. Cooley and J.W. Tukey, An algorithm for the machine calculation of complex Fourier series. *Math. Comput.* 19, pp. 297-301, 1965.

[9] J.J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson An extended set of FORTRAN Basic Linear Algebra Subprograms, *ACM Trans. Math. Soft.*, 14, pp. 1–17, 1988.

[10] J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling, A set of Level 3 Basic Linear Algebra Subprograms, *ACM Trans. Math. Soft.*, 16(1):1–17, 1990.

[11] E. Elmroth and F.G. Gustavson. Applying recursion to serial and parallel QR factorization leads to better performance. *IBM J. Res. Develop.*, 44(4):605–624, July 2000.

[12] 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(2):223–231, 1992.

[13] G. Golub, S. Nash, and C. Van Loan. A Hessenberg-Schur method for the matrix problem $AX + XB = C$. *IEEE Trans. Automat. Control*, AC-24:909–913, 1979.

[14] J. Gunnels, F.G. Gustavson, G. Henry, and R. van de Geijn. Formal linear algebra methods environment (FLAME). *ACM Trans. Math. Software*, 27(4):422–455, 2001.

[15] G.E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, 1965

[16] R. Granat, B. Kågström, and P. Poromaa. Parallel ScaLAPACK-Style Algorithms for Solving Continuous-Time Sylvester Matrix Equations. In Kosch et al. (eds), *Euro-Par 2003, Parallel Processing*, Lecture Notes in Computer Science, 2790:800–809, Springer-Verlag, 2003.

[17] F.G. Gustavson. New generalized data structures for matrices lead to a variety of high performance algorithms. In R.F. Boisvert and P.T.P. Tang, editors, *The Architectures for Scientific Software*, volume 188 of *IFIP Conference Proceedings*, pages 211–234. Kluwer, 2001.

[18] F. Gustavson, A. Henriksson, I. Jonsson, B. Kågström, and P. Ling. Recursive Blocked Data Formats and BLAS's for Dense Linear Algebra Algorithms. In Kågström et al. (eds), *Applied Parallel Computing, PARA'98*, Lecture Notes in Computer Science, 1541:195–206, Springer-Verlag, 1998.

[19] F. 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 Kågström et al. (eds), *Applied Parallel Computing, PARA'98*, Lecture Notes in Computer Science, 1541:207–217, Springer-Verlag, 1998.

[20] F. Gustavson, I. Jonsson, B. Kågström, and P. Ling. Peak Performance on Hierarchical Memory Architectures – New Recursive Blocked Data Formats and BLAS's. In Hendrickson et al. (eds), *Proceedings of the Ninth SIAM*

Conference on Parallel Processing for Scientific Computing, PPSC99, San Antonio, TX, USA. 1999. SIAM.

[21] F. Gustavson and I. Jonsson. High Performance Cholesky Factorization via Blocking and Recursion that uses Minimal Storage. In Bjorstad et al. (eds), *Applied Parallel Computing, New Paradigms for HPC Industry and Academia*, Lecture Notes in Computer Science, 1947, pp. 82-91, Springer-Verlag, 2000.

[22] W.W. Hager. Condition estimators. *SIAM J. Sci. Comput.*, 5:311–316, 1984.

[23] S.J. Hammarling. Numerical solution of the stable, non-negative definite Lyapunov equation. *IMA J. Num. Anal.*, 2:303–323, 1982.

[24] 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, SE-901 87 Umeå, 1998.

[25] N.J. Higham. Perturbation theory and backward error for $AX - XB = C$. *BIT*, 33:124–136, 1993.

[26] I. Jonsson and B. Kågström. Parallel Triangular Sylvester-Type Matrix Eqation Solvers for SMP Systems using Recursive Blocking. In Bjorstad et al. (eds), *Applied Parallel Computing, New Paradigms for HPC Industry and Academia*, Lecture Notes in Computer Science, 1947, pp. 64-73, Springer-Verlag, 2000.

[27] I. Jonsson and B. Kågström. Recursive Blocked Algorithms for Solving Triangular Matrix Equations—Part I: One-Sided and Coupled Sylvester-Type Equations, *SLICOT Working Note* 2001-4, 2001.

[28] I. Jonsson and B. Kågström. Recursive Blocked Algorithms for Solving Triangular Matrix Equations—Part II: Two-Sided and Generalized Sylvester and Lyapunov Equations, *SLICOT Working Note* 2001-5.

[29] I. Jonsson and B. Kågström. Parallel Two-Sided Sylvester-Type Matrix Equation Solvers for SMP Systems Using Recursive Blocking. In Kågström et al. (eds), *Applied Parallel Computing, PARA 2002*, Lecture Notes in Computer Science, 2367:297–306, Springer-Verlag, 2002.

[30] 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. Softw.*, 28(4):392–415, December 2002.

[31] I. Jonsson and B. Kågström. Recursive blocked algorithms for solving triangular systems — Part II: Two-sided and generalized Sylvester and Lyapunov matrix equations. *ACM Trans. Math. Softw.*, 28(4):416–435, December 2002.

[32] I. Jonsson and B. Kågström. Parallel Two-Sided Sylvester-Type Matrix Equation Solvers for SMP Systems using Recursive Blocking. In J. Fagerhom and et al, editors, *Applied Parallel Computing: Advanced Scientific Computing*, volume 2367, pages 297–306. Springer-Verlag, Lecture Notes in Computer Science, 2002.

[33] I. Jonsson and B. Kågström. RECSY — A High Performance Library for Sylvester-Type Matrix Equations. In Kosch et al. (eds), *Euro-Par 2003, Parallel Processing*, Lecture Notes in Computer Science, 2790:810–819, Springer-Verlag, 2003.

[34] B. Kågström. A perturbation analysis of the generalized Sylvester equation $(AR - LB, DR - LE) = (C, F)$. *SIAM J. Matrix Anal. Appl.*, 15(4):1045–1060, 1994.

[35] 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(3):268–302, 1998.

[36] 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(1):99–101, 1992.

[37] B. Kågström and L. Westin. Generalized Schur methods with condition estimators for solving the generalized Sylvester equation. *IEEE Trans. Automat. Control*, 34(4):745–751, 1989.

[38] C.L. Lawson, R.J. Hanson, D.Kincaid, and F.T. Krogh, Basic Linear Algebra Subprograms for FORTRAN usage, *ACM Trans. Math. Soft.*, 5, pp. 308–323, 1979

[39] T. Penzl. Numerical solution of generalized Lyapunov equations. *Advances in Comp. Math.*, 8:33–48, 1998.

[40] Valsalam V., and Skjellum A. A framework for high-performance matrix multiplication based on hierarchical abstractions, algorithms and optimized low-level kernels. *Concurrency and Computation: Practice and Experience*, 14(10):805–839

[41] SLICOT. The SLICOT library and the numerics in control network (NICONET) website. www.win.tue.nl/niconet/.

[42] V. Strassen. Gaussian elimination is not optimal. *Numerisch. Math.*, 13:354–356, 1969.

[43] Whaley, R., Petitet A., and Dongarra J. Automated Empirical Optimization of Software and the ATLAS Project. *Parallel Computing*, 27(1–2):3–25.

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

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