Scheduling of Parallel Matrix Computations and Data Layout Conversion for HPC and Multi-Core Architectures

Lars Karlsson

PhD Thesis, April 2011

Department of Computing Science
SE-901 87 Umeå

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

larsk@cs.umu.se

Copyright © 2011 by Lars Karlsson
Except Paper I, © ACM, 2009
Paper III, © F. G. Gustavson, L. Karlsson, and B. Kågström, 2011
Paper IV, © Springer, 2008
Paper V, © Springer, 2010
Paper VI, © Elsevier, 2011

ISBN 978-91-7459-214-6
ISSN 0348-0542
UMINF 11.04

Printed by Print & Media, Umeå University, 2011

Abstract

Dense linear algebra represents fundamental building blocks in many computational science and engineering applications. The dense linear algebra algorithms must be numerically stable, robust, and reliable in order to be usable as black-box solvers by expert as well as non-expert users. The algorithms also need to scale and run efficiently on massively parallel computers with multi-core nodes.

Developing high-performance algorithms for dense matrix computations is a challenging task, especially since the widespread adoption of multi-core architectures. Cache reuse is an even more critical issue on multi-core processors than on uni-core processors due to their larger computational power and more complex memory hierarchies. Blocked matrix storage formats, in which blocks of the matrix are stored contiguously, and blocked algorithms, in which the algorithms exhibit large amounts of cache reuse, remain key techniques in the effort to approach the theoretical peak performance.

In Paper I, we present a packed and distributed Cholesky factorization algorithm based on a new blocked and packed matrix storage format. High performance node computations are obtained as a result of the blocked storage format, and the use of look-ahead leads to improved parallel efficiency. In Paper II and Paper III, we study the problem of in-place matrix transposition in general and in-place matrix storage format conversion in particular. We present and evaluate new high-performance parallel algorithms for in-place conversion between the standard column-major and row-major formats and the four standard blocked matrix storage formats.

Another critical issue, besides cache reuse, is that of efficient scheduling of computational tasks. Many weakly scalable parallel algorithms are efficient only when the problem size per processor is relatively large. A current research trend focuses on developing parallel algorithms which are more strongly scalable and hence more efficient also for smaller problems.

In Paper IV, we present a framework for dynamic node-scheduling of two-sided matrix computations and demonstrate that by using priority-based scheduling one can obtain an efficient scheduling of a QR sweep. In Paper V and Paper VI, we present a blocked implementation of two-stage Hessenberg reduction targeting multi-core architectures. The main contributions of Paper V are in the blocking and scheduling of the second stage. Specifically, we show that the concept of look-ahead can be applied also to this two-sided factorization, and we propose an adaptive load-balancing technique that allow us to schedule the operations effectively.

Sammanfattning

Matrisberäkningar är fundamentala byggblock i många beräkningstunga teknisk-vetenskapliga applikationer. Algoritmerna måste vara numeriskt stabila och robusta för att användaren ska kunna förlita sig på de beräknade resultaten. Algoritmerna måste dessutom skala och kunna köras effektivt på massivt parallella datorer med noder bestående av flerkärniga processorer.

Det är utmanande att uveckla högpresterande algoritmer för täta matrisberäkningar, särskilt sedan introduktionen av flerkärniga processorer. Det är ännu viktigare att återanvända data i cache-minnena i en flerkärnig processor på grund av dess höga beräkningsprestanda. Två centrala tekniker i strävan efter algoritmer med optimal prestanda är blockade algoritmer och blockade matrislagringsformat. En blockad algoritm har ett minnesåtkomstmönster som passar minneshierarkin väl. Ett blockat matrislagringsformat placerar matrisens element i minnet så att elementen i specifika matrisblock lagras konsekutivt.

I Artikel I presenteras en algoritm för Cholesky-faktorisering av en matris kompakt lagrad i ett distribuerat minne. Det nya lagringsformatet är blockat och möjliggör därigenom hög prestanda. Artikel II och Artikel III beskriver hur en konventionellt lagrad matris kan konverteras till och från ett blockat lagringsformat med hjälp av en ytterst liten mängd extra lagringsutrymme. Lösningen bygger på en ny parallell algoritm för matristransponering av rektangulära matriser.

Vid skapandet av en skalbar parallell algoritm måste man även beakta hur de olika beräkningsuppgifterna schemaläggs på ett effektivt sätt. Många så kallade svagt skalbara algoritmer är effektiva endast för relativt stora problem. En nuvarande forskningstrend är att utveckla så kallade starkt skalbara algoritmer, vilka är mer effektiva även för mindre problem.

Artikel IV introducerar ett dynamiskt schemaläggningssystem för två-sidiga matrisberäkningar. Beräkningsuppgifterna fördelas statiskt på noderna och schemaläggs sedan dynamiskt inom varje nod. Artikeln visar även hur prioritetsbaserad schemaläggning tar en tidigare ineffektiv algoritm för ett så kallat QR-svep och gör den effektiv. Artikel V och Artikel VI presenterar nya parallella blockade algoritmer, designade för flerkärniga processorer, för en två-stegs Hessenberg-reduktion. De centrala bidragen i Artikel V utgörs av en blockad algoritm för reduktionens andra steg samt en adaptiv lastbalanseringsmetod.

Preface

This PhD Thesis consists of an introduction and the following six papers:

Paper I F. G. Gustavson, L. Karlsson, and B. Kågström. Distributed SBP Cholesky Factorization Algorithms with Near-Optimal Scheduling[1]. *ACM Transactions on Mathematical Software*, 36(2), pages 11:1–11:25, 2009.

Paper II L. Karlsson. Blocked In-Place Transposition with Application to Storage Format Conversion. Technical Report UMINF 09.01, Department of Computing Science, Umeå University, SE-901 87 Umeå, Sweden, January 2009.

Paper III F. G. Gustavson, L. Karlsson, and B. Kågström. Parallel and Cache-Efficient In-Place Matrix Storage Format Conversion. *ACM Transactions on Mathematical Software* (submitted).

Paper IV L. Karlsson and B. Kågström. A Framework for Dynamic Node-Scheduling of Two-Sided Blocked Matrix Computations[2]. In *Proceedings of PARA 2008*, LNCS. Springer, (to appear).

Paper V L. Karlsson and B. Kågström. Efficient Reduction from Block Hessenberg Form to Hessenberg Form using Shared Memory[3]. In *Proceedings of PARA 2010*, LNCS. Springer, (to appear).

Paper VI L. Karlsson and B. Kågström. Parallel Two-Stage Reduction to Hessenberg Form using Dynamic Scheduling on Shared-Memory Architectures[4]. *Parallel Computing*, (to appear).

[1] Reprinted by permission of ACM.
[2] Reprinted by permission of Springer.
[3] Reprinted by permission of Springer.
[4] Reprinted by permission of Elsevier.

Acknowledgments

First of all I would like to sincerely thank my advisor Professor Bo Kågström for his guidance and support. Bo, you have the admirable skill of being able to provide constructive feedback even at times when I have presented you with a very crude draft or a particularly confusing figure. Thank you!

Professor Fred Gustavson is a co-author to two of the papers in this thesis, and he has teached me, among many other things, the importance of blocked matrix layouts and recursive algorithms. Fred, I envy your attention to detail. Thank you!

PhD Robert Granat introduced me to the research area and encouraged me to apply for a PhD position. Robert also taught me parallel programming. Thank you!

I would also like to thank Professor Daniel Kressner and his wife Ana for kindly hosting me in Switzerland on several occasions. Thank you!

Thanks to all friends and colleagues at the CS department and HPC2N. In particular, thanks to the past and present members of the research group and my co-advisor Professor Erik Elmroth. Thanks for all the interesting discussions and conference trips. Special thanks to the support and administrative staffs at CS and HPC2N for keeping the machines and the organizations running smoothly.

For keeping my health under control, I would like to thank Anna, Eva, Ivan, and Ingrid as well as their past and present co-workers.

For their unconditional love and support, I wish to thank my entire family and my very best friend and fiancée Hannah. *Man ska inte göra det så lätt för sig!* ;)

Umeå, April 2011

Lars Karlsson

Contents

Chapter 1

Introduction

The subject of parallel matrix computations deals with the development and analysis of parallel and high-performance algorithms for a variety of different matrix computations, e.g., solving a linear system of equations, computing a linear least squares approximation, computing eigenvalues and eigenvectors, and computing singular value decompositions [37]. The emphasis is on developing stable, robust, and high-performance algorithms applicable to all scales from personal computers to massively parallel supercomputers with multi-core processors and accelerators. The aim is to produce reliable black-box solvers that can then be used as building blocks in established as well as future applications in, e.g., science and engineering. Hence the research is intentionally decoupled from any particular application and the end products are often disseminated in comprehensive proprietary as well as open source state-of-the-art software libraries [19, 84, 6, 44, 83, 73, 46, 45].

Since it is central to the topic, let us briefly introduce the concept of a matrix. An $m \times n$ matrix A is essentially a table with mn entries arranged into m rows and n columns. The entry on the i-th row and j-th column is denoted by a_{ij} or $A(i,j)$ or $(A)_{ij}$. A matrix is square if $m = n$ and it is rectangular otherwise. The transpose of an $m \times n$ matrix A, which we denote by A^T, is an $n \times m$ matrix $B = A^T$ such that $b_{ij} = a_{ji}$, i.e., the rows of A are the columns of A^T. For example,

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \qquad \text{and} \qquad A^T = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

illustrate a 2×3 rectangular matrix A and its 3×2 transpose A^T. A (square) matrix A is symmetric if $A^T = A$, in which case $a_{ij} = a_{ji}$.

Matrix addition and subtraction are defined component-wise for matrices of the same size. Matrices can also be multiplied together, but the definition of matrix multiplication is slightly more complicated than the definition of matrix addition. Specifically, if A and B are matrices of size $m \times p$ and $p \times n$,

respectively, then the matrix product $C = AB$ is an $m \times n$ matrix defined by the entries

$$c_{ij} = \sum_{k=1}^{p} a_{ik} b_{kj}.$$

Due to the associativity and commutativity of scalar addition and multiplication, it is possible to express matrix multiplication in different ways. In the setting of High-Performance Computing (HPC), block matrix multiplication plays a key role. A block matrix is an $M \times N$ matrix in which each entry, which we now call a block, is a matrix in itself. We illustrate this concept by a small example:

$$A = \left[\begin{array}{cc|cc|cc} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & a_{26} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & a_{36} \\ \hline a_{41} & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} & a_{56} \\ a_{61} & a_{62} & a_{63} & a_{64} & a_{65} & a_{66} \end{array}\right] = \left[\begin{array}{c|c|c} A_{11} & A_{12} & A_{13} \\ \hline A_{21} & A_{22} & A_{23} \end{array}\right].$$

In particular, we have the block

$$A_{12} = \begin{bmatrix} a_{13} & a_{14} \\ a_{23} & a_{24} \\ a_{33} & a_{34} \end{bmatrix}.$$

It turns out that the scalar definition of matrix multiplication extends to block matrices in a straightforward way. Specifically, let A and B be $M \times P$ and $P \times N$ block matrices, respectively. Then $C = AB$ is an $M \times N$ block matrix defined by the blocks

$$C_{ij} = \sum_{k=1}^{P} A_{ik} B_{kj}.$$

Block matrix multiplication is well defined if all the matrix multiplications that appear in the sum above are defined. In other words, the k-th block column of A must have the same number of columns as there are rows in the k-th block row of B.

The subject of matrix computations is loosely divided into the study of dense and sparse matrices. This division is motivated by the fact that a sparse matrix, i.e., a matrix in which a relatively large number of the entries are zero, must be treated with care in order to preserve its sparsity structure and thereby keep the computational cost and the storage requirements under control. In dense matrix computations, on the other hand, the matrix is either entirely filled in with non-zero elements or has a regular structure that is relatively easy to preserve, e.g., it may be upper/lower triangular, banded, symmetric, etc. Besides the algorithmic aspects of, e.g., complexity and reliability, important issues in the implementation of dense matrix computations are the minimization of communication, both with memory and with other processors, and effective load balancing across the processors.

HPC is a large and active research area that studies a broad range of topics related to the design and analysis of efficient and scalable parallel algorithms and parallel computers. Interest in HPC has recently increased significantly due to the shift from uni-core to multi-core processors and heterogeneous processors/systems [8]. As a consequence, parallel computing is now being applied on a large scale to mass-market applications in addition to its traditional applications in science and engineering.

Development of high-performance algorithms is challenging since in addition to the full set of problems related to the development of sequential programs, one must now also take into account the limitations and opportunities of the hardware on which the parallel programs are supposed to run. On the other hand, one often desires to have portability of both code and performance from one system to another. Thus, the conflicting goals of hardware awareness and portability implies that one should not tailor algorithms too much to a particular system but instead try to exploit whatever common features they might have. Fortunately, past experiences show that it often suffices to develop algorithms for a few broad classes of systems, e.g., shared memory, distributed memory, accelerators, and hybrids of these, and then adapt the resulting algorithms to particular systems by relying on optimized low-level libraries and/or tuning of algorithmic parameters.

Besides having algorithms that are reliable and robust, it is highly desirable that the algorithms can maintain a high efficiency when they are moved to a larger parallel computer. In general, if we increase the number of processors but keep the problem size constant, then the efficiency of a parallel algorithm tends to go down due to the fact that more processors also means that there are less data to store and process on each processor and hence the parallel overhead increases. An algorithm that maintains a high efficiency even when run on many processors relative to the problem size is said to be strongly scalable.

On the other hand, if we increase the problem size but keep the number of processors constant, then the efficiency tends to go up since the computational work assigned to each processor increases with the problem size and so the parallel overhead becomes easier to amortize. Scalable parallel algorithms can offset a drop in efficiency caused by moving to a larger parallel computer by also simultaneously increasing the problem size. Depending on how much the problem size must be increased to maintain a fixed efficiency, we can claim that one algorithm is more weakly scalable than another. Research has recently focused on developing strongly scalable algorithms, especially for multi-core architectures, in an effort to improve the efficiency of algorithms on a wide range of problem sizes that are not large enough to fully benefit from weak scalability.

The present thesis deals with a few important topics in parallel dense matrix computations. The targets include both distributed-memory systems and multi-core architectures.

Chapter 2

Principles of Parallel Dense Matrix Computations

The aim of this chapter is to introduce some fundamental principles of parallel dense matrix computations. See, e.g., [37, 43], for more detailed discussions.

2.1 Cache Reuse and Blocked Algorithms

High-performance algorithms have to deal with the relatively high latency and low bandwidth of the main memory. Processors are equipped with deep memory hierarchies, i.e., multiple small and fast so called cache memories located between the processor and the main memory, in order to reduce the main memory's negative effects on performance. The caches have lower latencies and higher bandwidths than the main memory and can therefore improve the effective latency and bandwidth. However, caches can only improve performance for certain types of memory access patterns. In particular, random memory accesses do not benefit at all from cache memories, whereas contiguous (or stride-1) memory accesses are ideal. Algorithms must reuse data stored in the caches multiple times in order to run efficiently. As we will see, this means reconsidering the design of both algorithms and data structures, since there are orders of magnitude differences in performance between the best and the worst memory access patterns.

The collection of fundamental linear algebra operations can be divided into three categories based on the type of objects that appear as inputs and outputs. Operations such as the dot product, $x^T y$, and the vector update or saxpy, $y \leftarrow \alpha x + y$, involve only vectors and are therefore called vector–vector operations (or Level 1 operations in the BLAS [20] terminology). Operations that involve both matrices and vectors are known as matrix–vector (Level 2) operations. Examples include the matrix-vector product, $y \leftarrow Ax$, and the outer product update, $A \leftarrow A - xy^T$. The third type involves only matrices and are called

matrix–matrix (Level 3) operations.

The operations in Level 1 and Level 2 perform only a few floating point operations (flops) per word of input/output, which means that they have limited cache reuse when the operands do not fit in cache. For example, the outer product update $A \leftarrow A - xy^T$, where A is an $m \times n$ matrix, requires $2mn$ flops and has $mn + m + n$ words of input/output. The Level 3 operations, on the other hand, benefit from a surface-to-volume effect and, depending on the shape and size of the matrices involved, can require many flops per word of input/output. For example, the Level 3 matrix multiplication operation $C \leftarrow C + AB$, where A, B, and C are $n \times n$ matrices, requires $2n^3$ flops but involves only $3n^2$ words of input/output. Moreover, this potential for cache reuse can be realized in practice, yielding high-performance implementations of the Level 3 operations capable of nearly attaining the theoretical peak performance.

If we count the number of bytes transferred to and from memory, b, as well as the number of flops, f, then the arithmetic intensity, defined as f/b, measures the amount of cache reuse and should ideally be larger than a system-dependent constant in order for the implementation to be efficient on that particular system.

The BLAS (Basic Linear Algebra Subprograms) is a widely adopted standard interface to a wide spectrum of operations covering the three levels discussed above [20, 61, 60, 15, 84, 10]. The idea behind the BLAS is to provide a standard interface to computational linear algebra kernels, which can then be optimized and tuned for particular computer architectures. The BLAS thereby enables portability of both code and performance for higher-level libraries. Among the many diverse operations in the BLAS, the matrix multiplication and add operation, `GEMM`, defined as $C \leftarrow \beta C + \alpha\,\mathrm{op}(A)\,\mathrm{op}(B)$ where $\mathrm{op}(X)$ is either X or its transpose X^T has a special status since all the other Level 3 operations can be expressed as mostly `GEMM` operations [61, 60]. Besides being universal, the `GEMM` operation is also highly regular and promotes reuse not only at the level of caches but also at the level of processor registers. In fact, `GEMM` is one of the few operations that map well enough to the features of a typical high-performance processor that the theoretical peak performance is within reach.

Based on the above considerations of cache reuse and portability, it follows that one should aim to express matrix computations in terms of `GEMM` and/or other Level 3 operations. Unfortunately, it turns out that some matrix computations are simpler to express in terms of Level 1 and Level 2 operations than in terms of Level 3 operations [43]. The key to obtaining Level 3 performance is often to formulate algorithms in terms of operations on matrix blocks. For this reason, high-performance linear algebra algorithms are often referred to as blocked algorithms. Moreover, some computations can be expressed recursively, which often yields an automatic multi-level blocking [47, 41]. Such algorithms are called cache-oblivious, since they automatically adapt to the memory hierarchy at run time [42].

The amount of cache reuse is influenced not only by the algorithms but

also by the data structures. This is due to the fact that caches transfer data in contiguous blocks, called cache lines, each of which can store dozens of (contiguously stored) matrix entries. Therefore, the amount of data that is actually transferred to/from a cache memory in response to a memory reference can be larger than the amount of data actually referenced. This means that the layout of matrices in memory (the so called matrix storage format or data layout) influences the cache reuse by determining the volume of transferred data. The optimal choice of data layout depends on the algorithm. In particular, since the memory access pattern of a blocked algorithm differs from that of an unblocked one, their optimal data layouts also differ. The standard row- and column-major data layouts, in which rows or columns of the matrix are stored contiguously, are suitable for unblocked algorithms, but blocked and recursive data layouts are more suitable for blocked algorithms [10, 9, 48, 41, 32, 31, 53, 5, 63, 70, 65].

In a distributed-memory environment, the matrices must be distributed across the processors such that a subset of the matrix entries are stored locally on each processor [37, 19]. A matrix distribution specifies for each entry of the matrix on which processor that entry should be stored. While one can conceive of many different matrix distribution schemes, only a few are of practical interest. The most common class of distributions assign a (not necessarily contiguous) sub-matrix, called the local sub-matrix, to each processor. The local sub-matrix is then stored using, e.g., the column-major data layout or some blocked data layout. A sub-matrix is defined by a subset of the columns and a (possibly different) subset of the rows. A distribution is said to be one-dimensional if one of these two subsets is equal to the entire set, and it is said to be two-dimensional otherwise.

The block column distribution is an example of a one-dimensional matrix distribution. Let us illustrate by distributing an 8×8 matrix A across four processors. The entries of A are assigned to the four processors (numbered from 0 to 3) as follows:

$$A = \left[\begin{array}{cc|cc|cc|cc} 0 & 0 & 1 & 1 & 2 & 2 & 3 & 3 \\ 0 & 0 & 1 & 1 & 2 & 2 & 3 & 3 \\ 0 & 0 & 1 & 1 & 2 & 2 & 3 & 3 \\ 0 & 0 & 1 & 1 & 2 & 2 & 3 & 3 \\ 0 & 0 & 1 & 1 & 2 & 2 & 3 & 3 \\ 0 & 0 & 1 & 1 & 2 & 2 & 3 & 3 \\ 0 & 0 & 1 & 1 & 2 & 2 & 3 & 3 \\ 0 & 0 & 1 & 1 & 2 & 2 & 3 & 3 \end{array}\right], \quad A_0 = \left[\begin{array}{cc} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \\ a_{41} & a_{42} \\ a_{51} & a_{52} \\ a_{61} & a_{62} \\ a_{71} & a_{72} \\ a_{81} & a_{82} \end{array}\right].$$

The entries in the local sub-matrix A_0 belonging to processor 0 is also shown above.

Even though one-dimensional distributions are simple, they unfortunately also limit the scalability of many algorithms [33]. Figure 1 illustrates one of the disadvantages of the block column distribution in particular and one-dimensional distributions in general. Figure 1 shows how the computational region of a typical factorization algorithm gradually shrinks toward the bottom

right corner of the matrix. Note in particular that the data stored on processor 0 is used only in the first of the four iterations in this example.

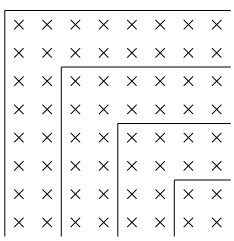


Figure 1: Progression of the computational region in (right-looking) factorization algorithms. The entire matrix is involved in the first iteration, but in subsequent iterations the computational region gradually shrinks toward the bottom right corner of the matrix.

Many distributed-memory matrix computations have better scalability if they are used in conjunction with a two-dimensional instead of a one-dimensional distribution. In particular, the block-cyclic distribution has proven to be a good compromise between load balance on the one hand and efficient communication patterns on the other hand. In the block-cyclic distribution, the matrix is first partitioned into blocks of some fixed size and then the blocks are distributed cyclically in both dimensions. Instead of providing a formal definition, let us illustrate by distributing an 8×8 matrix A on four processors:

$$A = \left[\begin{array}{ccc|ccc||cc} 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ \hline 2 & 2 & 2 & 3 & 3 & 3 & 2 & 2 \\ 2 & 2 & 2 & 3 & 3 & 3 & 2 & 2 \\ \hline\hline 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ \hline 2 & 2 & 2 & 3 & 3 & 3 & 2 & 2 \\ 2 & 2 & 2 & 3 & 3 & 3 & 2 & 2 \end{array}\right], \quad A_0 = \left[\begin{array}{ccc|cc} a_{11} & a_{12} & a_{13} & a_{17} & a_{18} \\ a_{21} & a_{22} & a_{23} & a_{27} & a_{28} \\ \hline a_{51} & a_{52} & a_{53} & a_{57} & a_{58} \\ a_{61} & a_{62} & a_{63} & a_{67} & a_{68} \end{array}\right].$$

As before, the matrix A_0 above denotes the local sub-matrix belonging to processor 0. Note that by using this two-dimensional distribution, the computational regions shown in Figure 1 become more evenly distributed.

The matrix is distributed in units of blocks instead of individual entries because of a trade-off between the cost of communication and load balance. In particular, the ScaLAPACK library uses the block-cyclic distribution with a generally non-unit block size [19]. The Elemental project [73], on the other hand, aims to develop a dense linear algebra library similar in functionality to ScaLAPACK, but based on a pure cyclic distribution (i.e., with unit block size).

2.2 Task Decomposition and Scheduling

In order to run on a parallel computer, a computation must be decomposed into tasks that can then be scheduled onto the processors. Both of these steps can be realized in many different ways. The aim of this section is to discuss those that are most relevant to the papers included in this thesis.

The task decomposition can either be determined statically (i.e., at compile time) or dynamically (i.e, at run time). In both cases, the task decomposition may or may not be dependent on the number of processors. Bit-wise reproducible results are difficult to guarantee unless the task decomposition is both static and independent of the number of processors. On the other hand, a dynamic task decomposition is useful if one wants to adaptively balance the load at run time.

In order to make the discussion more concrete, let us consider the following block outer product update:

$$C \leftarrow C - AB,$$

where C is $m \times n$ and A is $m \times k$ and B is $k \times n$. The cost of this block outer product is $2mnk$ flops. We introduce parallelism by partitioning the rows of A and the columns of B into blocks. Specifically, we partition the m rows of A into r blocks of size $m_1, m_2, \ldots, m_r$ and the n columns of B into c blocks of size $n_1, n_2, \ldots, n_c$. For example, if we set $r = 3$ and $c = 2$ we obtain $rc = 6$ independent tasks of the form $C_{ij} - A_i B_j$, each with a cost of $2m_i n_j k$ flops, since

$$C = C - AB = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \\ C_{31} & C_{32} \end{bmatrix} \leftarrow \begin{bmatrix} C_{11} - A_1 B_1 & C_{12} - A_1 B_2 \\ C_{21} - A_2 B_1 & C_{22} - A_2 B_2 \\ C_{31} - A_3 B_1 & C_{32} - A_3 B_2 \end{bmatrix}.$$

If we set the block sizes such that $m_i = n_j = K$ for some constant K, then the tasks have a uniform size (i.e., $2K^2k$ flops) and the decomposition is static and independent of the number of processors. If we instead set the number of blocks such that $r = c = \sqrt{p}$, and set the block sizes such that $m_i = m/\sqrt{p}$ and $n_j = n/\sqrt{p}$, where p is the number of processors, then the tasks are again uniformly sized and the decomposition is static, but now it depends on the number of processors. Lastly, if we make the choice regarding the number of blocks r and c and/or the block sizes m_i and n_j at run time, then we get a dynamic task decomposition.

The granularity of the tasks is an important consideration since not only does it affect the number of tasks and hence the scheduling overhead, but it also affects the performance of the tasks. A fine-grained decomposition exposes a lot of parallelism but makes it difficult to perform each task efficiently. In contrast, a coarse-grained decomposition exposes less parallelism but improves the performance of each task. In other words, there is a trade-off between the amount of exposed parallelism and the performance of each task. Finding a suitable balance can be tricky, and this issue is explored in, e.g., [3].

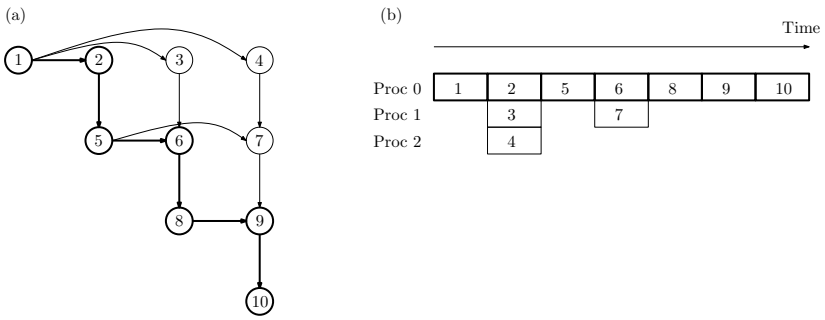


Figure 2: (a) An example task graph (DAG) with uniform processing times, and (b) an optimal scheduling of the task graph onto three processors.

Once the tasks are known they can be scheduled onto the available processors. The order in which the tasks can be run is often constrained by precedence constraints, each of which says that one task must finish before another one can start. The scheduling of tasks must of course respect these constraints in order for the computation to yield correct results. A common way to model tasks with precedence constraints is as a task graph, i.e., as a Directed Acyclic Graph (DAG), where the nodes represent tasks and the edges represent precedence constraints. For example, consider the task graph in Figure 2(a). We see that we have to start with task 1 since all other tasks depend on it either directly or indirectly. Then we can choose between tasks 2–4 or even do them all in parallel, and so on. If we assume that the tasks have uniform processing times, then the schedule shown in Figure 2(b) is optimal in the sense that it has minimal length, since any schedule must respect the linear order imposed by the critical path $1 \rightarrow 2 \rightarrow 5 \rightarrow 6 \rightarrow 8 \rightarrow 9 \rightarrow 10$, which is highlighted in Figure 2(a). A path with maximal length, where the length of a path is defined as the sum of the processing times, is said to be a critical path. A critical path imposes a lower bound on the parallel execution time since the tasks on any path must be performed in sequence. Unfortunately, it turns out to be very expensive to find an optimal schedule even with such an unrealistic assumption as uniform processing times. In reality, processing times are often noisy, difficult to model accurately, and due to the presence of caches they might even depend on the schedule. For these reasons, we are forced to use heuristics when scheduling matrix computations.

Let us consider the parallelization of the one-sided matrix factorizations LU, Cholesky, and QR in order to introduce the important concept of look-ahead. In their right-looking variants, one-sided factorization algorithms can be structured as an outer loop where each iteration has two parts: First a "panel factorization", which applies the factorization to a narrow block of columns (a panel), followed by a "trailing matrix update", which updates the trailing matrix based on the previously factored panel. Let us map each of these operations to a task. Figure 3(a) illustrates the resulting task graph. Note that the

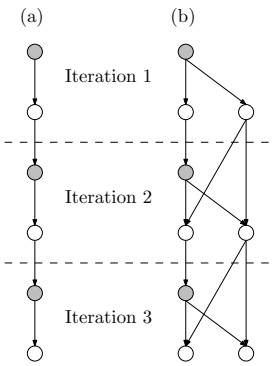


Figure 3: Illustration of the look-ahead concept in factorization algorithms. "Panel factorizations" are gray nodes, and "trailing matrix updates" are white nodes.

dependencies force us to execute the tasks in sequence, and, in particular, we cannot overlap "panel factorizations" with "trailing matrix updates". This limits the strong scalability, especially on multi-core architectures, since "panel factorizations" are more difficult to parallelize and often perform worse than the "trailing matrix updates".

Fortunately, it turns out that the next "panel factorization" does not depend on all of the current "trailing matrix update" [34]. Indeed, we only need to update the next panel in order to proceed. Let us therefore refine the task decomposition in Figure 3(a) by splitting the "trailing matrix update" in two: One task for the updating of the next panel and another task for the updating of the rest of the trailing matrix. Figure 3(b) illustrates the corresponding refined task graph. Note that the new task decomposition allows us to overlap the next "panel factorization" with (most of) the current "trailing matrix update". This technique of performing "panel factorizations" early is known as look-ahead and is a key to improving the strong scalability of factorization algorithms [27, 50, 57].

The task decompositions in Figure 3 expose too little parallelism and we need to further decompose the update tasks into subtasks. We begin by considering the scheduling of a basic factorization algorithm without look-ahead. Since the "panel factorizations" do not parallelize as well as the "trailing matrix updates", we assign them to a single processor. We can, however, parallelize the "trailing matrix updates" by, e.g., distributing its subtasks evenly across all processors. Figure 4(a) schematically illustrates an execution trace based on this task decomposition. There are four processors in this example, each one shown on its own row in the trace. Note that while the first processor is performing a "panel factorization", all other processors are idly waiting for it to finish, thereby wasting resources.

Let us now consider the scheduling of a factorization algorithm with look-ahead. For the same reasons as before, we assign the "panel factorizations"

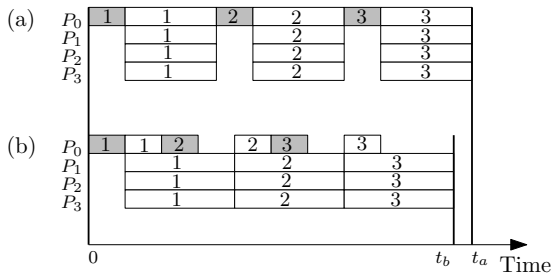


Figure 4: Schematic illustrations of execution traces for (a) a basic factorization algorithm, and (b) a factorization algorithm with look-ahead. A "panel factorization" is a gray box, and a "trailing matrix update" is a white box. The number within a box indicates the iteration to which that task belongs. For clarity, only the first three iterations are illustrated.

to a single processor. We also assign the (small) update of the next panel to that same processor. However, we distribute the rest of the "trailing matrix updates" evenly across the remaining processors. Figure 4(b) illustrates an execution trace based on this task decomposition. Note that after the first "panel factorization", all processors except the first one are busy until the very end. The first processor, however, is idle some of the time since its work load is not balanced with respect to the others.

Figure 4(b) suggests that we can improve the utilization of the first processor by assigning to it also a small portion of the rest of the "trailing matrix updates". However, we should not assign more work than what is necessary to eliminate the gaps in the schedule. If we want the subtasks of a "trailing matrix update" to be coarse-grained, e.g., one subtask per processor, then we must determine a priori how much work to assign to each processor in order to balance the load. Therefore, we need to model the execution time of the tasks on the first processor somehow. In Paper V [57], we use this technique, in conjunction with a model that is adapted at run time, in an effort to obtain balanced coarse-grained tasks in the context of a two-sided factorization algorithm with look-ahead.

Another way to improve the utilization of the first processor is to partition the "trailing matrix update" into tasks with finer granularity and use dynamic scheduling, e.g., via a (centralized or decentralized) task pool [62, 64, 29, 11, 74, 72]. This approach has the advantage that there is no need to model the execution time of any of the tasks. However, a fine-grained decomposition might reduce the performance of local computations too much.

Figure 4(b) also suggests that the sequence of dependent tasks assigned to the first processor restricts the strong scalability of the algorithm. Indeed, once the number of processors reaches a certain critical point, the first processor will be active throughout the entire execution and impose a lower bound on the overall execution time. Partly motivated by this scalability problem, so

called tiled algorithms further decompose also the tasks assigned to the first processor in an effort to extract more parallelism and reduce the length of the critical paths in the task graph [74, 11, 29, 1, 67, 66, 23, 22, 80, 52, 2, 28, 64, 62, 63, 27, 30].

A natural extension of the idea to overlap the execution of different operations, e.g., "panel factorization" and "trailing matrix update", within an algorithm is to overlap tasks from one algorithm with tasks from another algorithm. To make the implementation of this idea practical, one probably needs to rely on some sort of dynamic scheduling framework [2, 51, 21, 71, 30].

Hybrid systems consisting of several multi-core processors connected to one or more computational accelerators (e.g., GPUs) present new challenges with respect to task decomposition and load balancing. See, e.g., [7, 12, 13, 14, 36, 76] for some recent developments in this direction.

Chapter 3

Background

The aim of this chapter is to provide some background information on the various topics discussed in the included papers.

3.1 Householder Reflections

At the core of dense matrix computations lies a collection of matrix factorizations. Examples include the one-sided matrix factorizations LU, Cholesky, and QR as well as the two-sided reductions to the Hessenberg, bidiagonal, and tridiagonal forms. The two-sided reductions are typically based on orthogonal transformations because of their favorable numerical stability properties. Paper V and Paper VI rely on a specific type of orthogonal transformation called Householder reflections, which we review next.

A Householder reflection is an orthogonal matrix of the form

$$Q = I - \frac{2}{v^T v} v v^T, \tag{3.1}$$

where $v \in \mathbb{R}^n$ is some non-zero vector [43]. By choosing v in a certain way, we can map a given (non-zero) n-vector x to a multiple of a given (non-zero) n-vector y. Specifically, given x and y we can construct Q of the form (3.1) such that $Qx = \pm(\|x\|_2/\|y\|_2)y$. In particular, if we choose $y = e_1$, where $e_1 = \begin{bmatrix} 1 & 0 & \cdots & 0 \end{bmatrix}^T$, then Q zeros all but the first entry in the vector x [43].

Let us show how to find the Householder vector v given the vectors x and y. We have

$$Qx = x - \frac{2v^T x}{v^T v} v,$$

and since we desire to have Qx as a multiple of y, we must have v as a linear combination of x and y. Assuming that $v = x + \alpha y$, we get

$$v^T x = x^T x + \alpha x^T y \qquad \text{and} \qquad v^T v = x^T x + 2\alpha x^T y + \alpha^2 y^T y.$$

Consequently, we can write Qx as

$$Qx = \left(1 - 2\frac{x^T x + \alpha x^T y}{x^T x + 2\alpha x^T y + \alpha^2 y^T y}\right) x - \left(2\alpha \frac{v^T x}{v^T v}\right) y,$$

i.e., we can express Qx as a linear combination of x and y. In order for Qx to be a multiple of y, we must set the coefficient of x to zero. We hence get

$$\alpha = \pm \frac{\|x\|_2}{\|y\|_2} \qquad \text{and so} \qquad v = x \pm \frac{\|x\|_2}{\|y\|_2} y.$$

In the important special case when $y = e_1$, we have $\|y\|_2 = 1$ and so we can find v simply by adding or subtracting the norm of x from the first entry of x.

Note that we need n parameters, i.e., the n components of v, in order to annihilate $n - 1$ entries. If we overwrite the annihilated entries with $n - 1$ of the parameters, then we need one additional word of storage to store the last parameter. The LAPACK convention is to scale v such that $v_1 = 1$. The scaling factor is stored separately. Thus in practice, a Householder reflection is often represented in the form

$$Q = I - \tau v v^T \qquad \text{where} \quad \tau = \frac{2}{v^T v} \quad \text{and} \quad v_1 = 1,$$

with the entries v_2 thru v_n stored in the entries annihilated by the reflection and the scaling factor τ stored in an additional word of storage. We have already seen that Householder reflections are cheap to compute. They are also cheap to apply. Indeed, the cost of applying a Householder reflection $Q = I - \tau v v^T$ to an n-vector x is approximately $4n$ flops if we perform the update as follows:

$$x \leftarrow Qx = x - v(\tau(v^T x)).$$

Applying a Householder reflection to a matrix is essentially a Level 2 operation, i.e., a matrix–vector operation, and is therefore slow if the matrix does not fit in cache. As it turns out, many algorithms apply not just one but a product of several Householder reflections. Such products can be expressed in a format known as the (compact) WY representation and applied using Level 3 operations [18, 78, 43, 54].

The compact WY representation can be defined recursively as follows. If $Q = I - \tau v v^T$ is a Householder reflection, then $Q = I - VTV^T$, where $V = v$ and $T = \tau$, is its compact WY representation. If we let $I - V_1 T_1 V_1^T$ denote the compact WY representation of the product $Q_1 Q_2 \cdots Q_{r_1}$ and let $I - V_2 T_2 V_2^T$ denote the compact WY representation of the product $Z_1 Z_2 \cdots Z_{r_2}$, then

$$Q_1 \cdots Q_{r_1} Z_1 \cdots Z_{r_2} = (I - V_1 T_1 V_1^T)(I - V_2 T_2 V_2^T) = I - VTV^T$$

is the compact WY representation of their product, where

$$V = \begin{bmatrix} V_1 & V_2 \end{bmatrix} \qquad \text{and} \qquad T = \begin{bmatrix} T_1 & -T_1 V_1^T V_2 T_2 \\ 0 & T_2 \end{bmatrix}.$$

The computation and application of the upper triangular matrix T requires some additional flops, but this extra cost is negligible if the order n of the Householder reflections is much larger than the number of reflections in the product.

3.2 Hessenberg Decomposition

A matrix H is in (upper) Hessenberg form if the entries of H are zero below its first sub-diagonal [82, 43]. For example, the following matrix is in Hessenberg form:

$$H = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ 0 & 0 & \times & \times & \times \\ 0 & 0 & 0 & \times & \times \end{bmatrix},$$

where $\times$ denotes a (possibly) non-zero entry. Any square $n \times n$ real matrix $A \in \mathbb{R}^{n \times n}$ can be reduced to Hessenberg form by an orthogonal similarity transformation, i.e., a transformation of the type

$$A \leftarrow Q^T A Q \qquad \text{where} \qquad Q^T Q = I.$$

We now describe the standard one-stage Hessenberg reduction algorithm [43], which is based on Householder reflections. We begin by partitioning A:

$$A = \begin{bmatrix} a_{11} & a_{12}^T \\ a_{21} & A_{22} \end{bmatrix}.$$

Then we construct a Householder reflection Q_1 of order $n-1$ such that $Q_1^T a_{21}$ becomes a multiple of e_1. Next, we update A via the similarity transformation

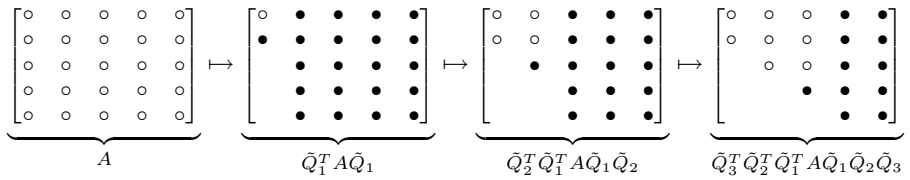
$$A \leftarrow \begin{bmatrix} a_{11} & a_{12}Q_1 \\ Q_1^T a_{21} & Q_1^T A_{22} Q_1 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12}Q_1 \\ \pm\|a_{21}\|_2 e_1 & Q_1^T A_{22} Q_1 \end{bmatrix}.$$

Note that A is now in Hessenberg form in its first column since only the first entry of $Q_1^T a_{21}$ is non-zero. We can apply the same procedure to reduce each column in turn from left to right and thereby reduce the entire matrix to Hessenberg form using a total of $n-2$ reflections of order $n-1, n-2, \ldots, 2$. In summary, the algorithm computes a Hessenberg matrix H such that

$$H = Q^T A Q, \qquad \text{where} \quad Q = \tilde{Q}_1 \tilde{Q}_2 \cdots \tilde{Q}_{n-2} \quad \text{and} \quad \tilde{Q}_k = \begin{bmatrix} I_k & \\ & Q_k \end{bmatrix}.$$

Let us illustrate the transitions from dense matrix to Hessenberg form on a

5×5 matrix:



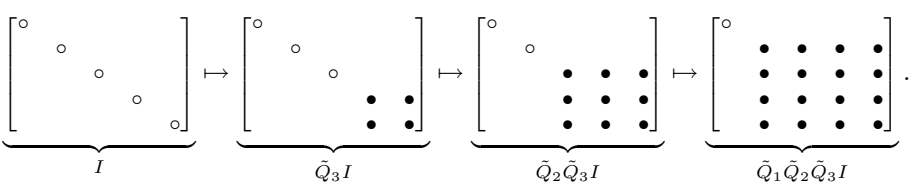
The non-zero entries are indicated above by a circle ($\circ$) and the recently modified non-zero entries are indicated by a bullet ($\bullet$). Note that this two-sided algorithm is another example where the computational region shrinks during the computation (see Figure 1 and Section 2.1) and so a two-dimensional distribution of the matrix is preferred for distributed-memory computing.

The standard one-stage Hessenberg reduction algorithm requires $(10/3)n^3 + \mathcal{O}(n^2)$ flops if we are satisfied with having Q returned in factored form. If we want to compute Q explicitly, then the reflections can be accumulated efficiently in reverse order, i.e., by computing

$$Q = \tilde{Q}_1(\cdots(\tilde{Q}_{n-3}(\tilde{Q}_{n-2}I))\cdots).$$

This backward accumulation scheme is cheaper than forward accumulation since the initial sparsity of the identity matrix is preserved far longer if we apply the reflections in reverse order [43].

We illustrate the progression of the non-zero entries in the backward accumulation scheme by a 5×5 example:



The cost of explicitly computing Q via the backward accumulation scheme is $(4/3)n^3 + \mathcal{O}(n^2)$ flops, i.e., an additional 40% compared to only computing H.

The cache-efficiency of the algorithm above can be improved by casting some of the flops in terms of Level 3 operations through the (compact) WY representation [18, 78, 38, 75]. However, the best known formulation, i.e., [75], still performs 20% of the flops as large matrix–vector multiplications, which is a slow Level 2 operation. One way around this problem is to use hybrid CPU/GPU computing [81]. Another way is to use a two-stage reduction algorithm, which we describe next.

In the two-stage algorithm, the input matrix is first reduced to a block Hessenberg form with $r > 1$ non-zero sub-diagonals, and then the block Hessenberg form is reduced to actual Hessenberg form using a bulge-chasing procedure similar to the algorithms used for symmetric band reduction [16, 17, 79, 77, 68].

The first stage (Stage 1) can be realized in a manner similar to the one-stage algorithm, only with Householder reflections of a slightly smaller order. The Stage 1 algorithm does not, however, inherit the performance problems of the one-stage algorithm and is therefore quite efficient [58, 62]. The second stage (Stage 2) is very different from Stage 1, so let us describe it in some detail.

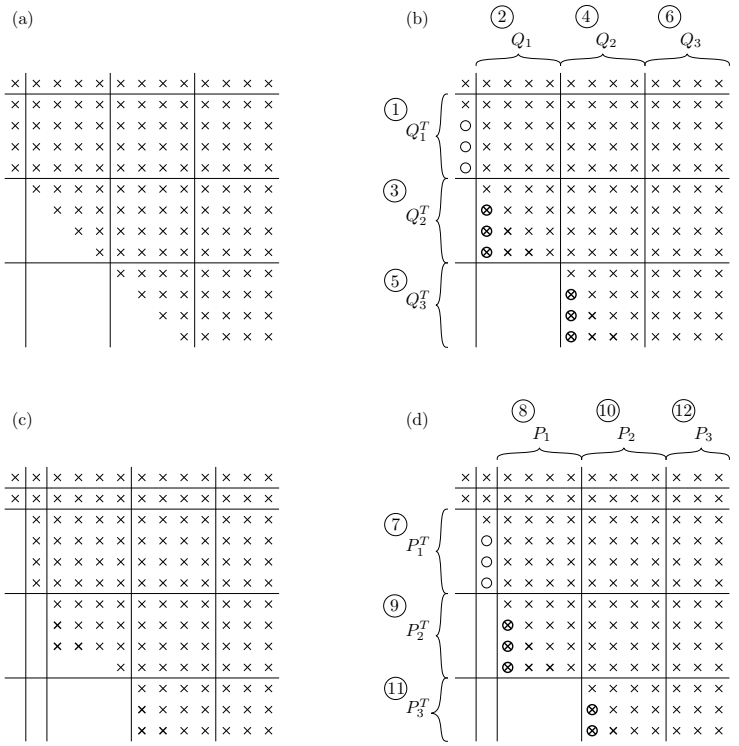


Figure 5: Illustration of the first two sweeps of Stage 2. (a) Input matrix in block Hessenberg form with $r = 4$ non-zero sub-diagonals, (b) the first sweep, (c) the input to the second sweep, and (d) the second sweep.

The basic idea for Stage 2 is to reduce the columns from left to right using Householder reflections of (at most) order r. Consider Figure 5(a), which illustrates a block Hessenberg matrix with $r = 4$ non-zero sub-diagonals. In the first sweep (i.e., iteration), the first column is reduced by a Householder reflection Q_1 as illustrated by Step 1 in Figure 5(b). When Q_1 is applied on the right-hand side (Step 2) in order to complete a similarity transformation, a bulge consisting of unwanted non-zero entries appears in the strictly lower triangular part of the sub-diagonal block in position $(3, 2)$. The fill-in entries are shown as bold crosses ($\times$) in Figure 5. The next step is to reduce the first column of the newly created bulge using a second reflection, Q_2, applied to the left-hand side in Step 3. But when Q_2 is applied to the right-hand side

in Step 4, a second bulge appears further down the diagonal in block $(4, 3)$. The process ends at the bottom right corner of the matrix. In this example, the first sweep is completed by reducing the first column of the second bulge (Steps 5–6).

The result of the first sweep is illustrated in Figure 5(c) with a slightly modified block partitioning. The second sweep is completely analogous to the first and is illustrated in Figure 5(d). Note in particular that the partially reduced bulges that were left by the first sweep align with the bulges created in the second sweep. This explains why one should reduce only the first column and not the entire bulge.

For a fixed value of r, the cost of Stage 1 is $(10/3)n^3 + \mathcal{O}(n^2)$ flops for the reduction to block Hessenberg form and an additional $(4/3)n^3 + \mathcal{O}(n^2)$ flops for the explicit formation of the orthogonal transformation matrix. Hence, the cost of Stage 1 alone is comparable to the cost of the entire one-stage reduction algorithm. Moreover, the cost of Stage 2 is non-negligible. Indeed, Stage 2 requires $2n^3 + \mathcal{O}(n^2)$ flops for the reduction (bulge-chasing) to Hessenberg form and an additional $2n^3 + \mathcal{O}(n^2)$ flops for explicit accumulation of the reflections. Interestingly, the number of sub-diagonals, r, has only a minor impact on the cost of the two-stage algorithm provided that $r \ll n$.

	One-stage		Two-stage	
	H	H and Q	H	H and Q
# flops	$(10/3)n^3$	$(14/3)n^3$	$(16/3)n^3$	$(26/3)n^3$

Table 3.1: The number of flops required for the one-stage and two-stage algorithms with and without explicit formation of the orthogonal transformation matrix.

The number of flops required for the one-stage and two-stage algorithms with and without explicit accumulation of the orthogonal transformation matrix are summarized in Table 3.1. However, these flop counts hide the fact that some flops can be applied more efficiently than others. In Paper V and Paper VI, we describe a novel implementation of the two-stage algorithm on multi-core architectures and show that despite requiring many more flops, the two-stage algorithm can be made to run much faster than the one-stage algorithm since it allows for more cache reuse.

3.3 Cholesky Factorization

A square $n \times n$ real matrix $A \in \mathbb{R}^{n \times n}$ is said to be symmetric positive definite if $A = A^T$ (i.e., A is symmetric) and the inequality $x^T A x > 0$ holds for all nonzero real vectors $0 \neq x \in \mathbb{R}^n$ (i.e., A is positive definite). Matrices which are symmetric positive definite arise naturally in many applications. For instance, if B has full column rank, then the matrix $A = B^T B$ is symmetric positive

definite because $A^T = (B^T B)^T = B^T B = A$ and $0 \neq x \in \mathbb{R}^n$ implies

$$x^T A x = x^T (B^T B) x = (Bx)^T (Bx) = \|Bx\|_2^2 > 0.$$

If A is symmetric positive definite, then the linear system $Ax = b$ can be stably solved by computing the so called Cholesky factorization $A = LL^T$ of A, where L is a unique lower triangular matrix with positive diagonal entries [43]. After factorizing A, one finds the solution $x = A^{-1}b = L^{-T}L^{-1}b$ by solving first the lower triangular system $Ly = b$ for y and then the upper triangular system $L^T x = y$ for x. There are many different algorithms for computing the Cholesky factorization of a dense matrix. Let us derive a recursive algorithm. We begin by partitioning A into a 2×2 block matrix

$$A = \begin{bmatrix} B & C^T \\ C & D \end{bmatrix} \tag{3.2}$$

such that both B and D are square. By equating corresponding blocks in the equation

$$A = \begin{bmatrix} B & C^T \\ C & D \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} L_{11}^T & L_{21}^T \\ 0 & L_{22}^T \end{bmatrix} = LL^T \tag{3.3}$$

we obtain, in particular, the three block equations

$$B = L_{11}L_{11}^T, \quad C = L_{21}L_{11}^T, \quad \text{and} \quad D - L_{21}L_{21}^T = L_{22}L_{22}^T. \tag{3.4}$$

Since L_{11} is lower triangular with positive diagonal entries, it has full row rank and so $L_{11}L_{11}^T$ (and hence B) is symmetric positive definite and L_{11} must therefore be the Cholesky factor of B. Thus, we can compute L_{11} by recursively factorizing B. Now that we have computed the block L_{11}, we can also compute the block $L_{21} = CL_{11}^{-T}$ by solving a triangular system with multiple right-hand sides. The last remaining block of L, i.e., the block L_{22}, also has full row rank since it is lower triangular with positive diagonal entries and so $D - L_{21}L_{21}^T$ must be symmetric positive definite and L_{22} its Cholesky factor, which we compute recursively. Algorithm 1 formally presents the steps described above.

Algorithm 1 Recursive Cholesky factorization: $L = \text{chol}(A)$

1: **if** A is 1×1 **then**
2: $A = \alpha > 0$, so return $L = \sqrt{\alpha}$.
3: **else**
4: Partition $A = \begin{bmatrix} B & C^T \\ C & D \end{bmatrix}$ such that both B and D are square.
5: Recursively compute $L_{11} \leftarrow \text{chol}(B)$.
6: Compute $L_{21} = CL_{11}^{-T}$ by solving the triangular system $L_{21}L_{11}^T = C$.
7: Update $D \leftarrow D - L_{21}L_{21}^T$.
8: Recursively compute $L_{22} \leftarrow \text{chol}(D)$.
9: Return $L = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix}$.
10: **end if**

Algorithm 1 requires $(1/3)n^3+\mathcal{O}(n^2)$ flops. We can reformulate Algorithm 1 using a loop instead of recursion by replacing the recursive call on line 5 with a call to a non-recursive Cholesky factorization algorithm, and replacing the tail-recursive call on line 8 with a loop. The resulting non-recursive algorithm is referred to as the "right-looking" variant of Cholesky factorization because it reads the right-most parts of A during the computation of $D - L_{21}L_{21}^T$ on line 7. The right-looking variant is suitable for parallel computing, especially on distributed-memory systems, since the operation on line 7 is easy to parallelize [33].

3.4 Matrix Storage Formats

As we have briefly touched upon already, the cache reuse is determined in part by the algorithm and in part by the data layout. The aim of this section is to introduce a few dense matrix storage formats. We temporarily switch to a zero-based index convention in order to simplify the notation.

If we store the $m \times n$ matrix A in the two-dimensional C array `A[m][n]` using the convention that

$$\texttt{A[}i\texttt{][}j\texttt{]} = a_{ij},$$

then the C language standard tell us that the entry a_{ij} is stored at offset $in+j$ from the base address of the array `A`. Hence the rows of the matrix A are stored contiguously, and for this reason we refer to

$$(i, j) \mapsto in + j$$

as the row-major storage mapping. We can also store the matrix A in the array `B[n][m]` (note the opposite order of m and n) using the convention that

$$\texttt{B[}j\texttt{][}i\texttt{]} = a_{ij}.$$

Now the columns instead of the rows of the matrix A are stored contiguously and we therefore refer to

$$(i, j) \mapsto jm + i$$

as the column-major storage mapping. The column-major storage format is used in, e.g., the BLAS [20], LAPACK [6], and ScaLAPACK [19] libraries.

Since the row- and column-major storage formats are not ideally suited for blocked matrix computations, we would rather like to use blocked storage formats. Suppose that we can factor $m = Mm_b$ and $n = Nn_b$. Then we can also partition A into an $M \times N$ block matrix

$$A = \begin{bmatrix} A_{0,0} & \cdots & A_{0,N-1} \\ \vdots & & \vdots \\ A_{M-1,1} & \cdots & A_{M-1,N-1} \end{bmatrix}$$

such that each block $A_{i,j}$ has size $m_b \times n_b$. We trivially have

$$(A_{i_2,j_2})_{i_1,j_1} = a_{ij} \qquad \text{where} \quad i = i_2 m_b + i_1 \quad \text{and} \quad j = j_2 n_b + j_1.$$

Thus, we can identify the entry a_{ij} either by the pair (i, j) or by the quadruple (i_2, i_1, j_2, j_1) defined implicitly by the relations above. Using the quadruple representation, we can store A in the row-major format using the (non-trivial) four-dimensional C array `AA[`M`][`m_b`][`N`][`n_b`]` with the convention that

$$\texttt{AA[}i_2\texttt{][}i_1\texttt{][}j_2\texttt{][}j_1\texttt{]} = (A_{i_2,j_2})_{i_1,j_1} = a_{ij}.$$

The corresponding offset from the base address of `AA` is

$$i_2 m_b N n_b + i_1 N n_b + j_2 n_b + j_1 = (i_2 m_b + i_1)n + (j_2 n_b + j_1) = in + j,$$

which shows that the array `AA` stores the matrix A in the row-major format.

Just as we can reorder the dimensions of the array `A` to go from the row-major storage format to the column-major storage format (i.e., to the array `B`), we can similarly reorder the dimensions of the four-dimensional array `AA` and go to $4! = 24$ different storage formats. Two of these formats correspond to the row- and column-major storage formats. In addition, four of the formats correspond to what we call the standard blocked storage formats, each of which stores the blocks of A contiguously and use the row- and/or column-major storage formats to lay out the blocks and the entries within each block. The four blocked storage formats correspond to arrays in which the dimensions M and N (in any order) come before the dimensions m_b and n_b (in any order). Specifically, we have the following array representations of the four blocked storage formats:

$$\begin{aligned}
&\texttt{CCRB[}N\texttt{][}M\texttt{][}n_b\texttt{][}m_b\texttt{]}, && \texttt{CCRB[}j_2\texttt{][}i_2\texttt{][}j_1\texttt{][}i_1\texttt{]} = a_{ij},\\
&\texttt{CRRB[}N\texttt{][}M\texttt{][}m_b\texttt{][}n_b\texttt{]}, && \texttt{CRRB[}j_2\texttt{][}i_2\texttt{][}i_1\texttt{][}j_1\texttt{]} = a_{ij},\\
&\texttt{RCRB[}M\texttt{][}N\texttt{][}n_b\texttt{][}m_b\texttt{]}, && \texttt{RCRB[}i_2\texttt{][}j_2\texttt{][}j_1\texttt{][}i_1\texttt{]} = a_{ij},\\
&\texttt{RRRB[}M\texttt{][}N\texttt{][}m_b\texttt{][}n_b\texttt{]}, && \texttt{RRRB[}i_2\texttt{][}j_2\texttt{][}i_1\texttt{][}j_1\texttt{]} = a_{ij}.
\end{aligned}$$

Let us also add the row- and column-major formats to the list:

$$\begin{aligned}
&\texttt{CM[}N\texttt{][}n_b\texttt{][}M\texttt{][}m_b\texttt{]}, && \texttt{CM[}j_2\texttt{][}j_1\texttt{][}i_2\texttt{][}i_1\texttt{]} = a_{ij},\\
&\texttt{RM[}M\texttt{][}m_b\texttt{][}N\texttt{][}n_b\texttt{]}, && \texttt{RM[}i_2\texttt{][}i_1\texttt{][}j_2\texttt{][}j_1\texttt{]} = a_{ij}.
\end{aligned}$$

In Paper II [55] and Paper III [51], we study the problem of rearranging the entries of any of these six arrays in-place, i.e., using only a constant amount of additional storage, such that the new arrangement corresponds to one of the other arrays. In Paper III [51], we develop new parallel storage format conversion algorithms targeted for multi-core architectures.

The storage formats that we have discussed so far are aimed at storing a full dense matrix. However, a significant amount of storage can be saved if,

for instance, the matrix is triangular or symmetric since only about half of the matrix entries need to be stored explicitly. We refer to storage formats that exploit the structure of the matrix to save space as packed storage formats.

The so called Square Block Packed (SBP) storage format [50] is applicable to triangular and symmetric matrices. This packed format enables Level 3 performance and is suitable for tiled algorithms. We assume without loss of generality that the matrix A is $n \times n$ and lower triangular. Given a block size n_b, we partition A into a block matrix

$$A = \begin{bmatrix} A_{0,0} & \cdots & A_{0,N-1} \\ \vdots & & \vdots \\ A_{N-1,0} & \cdots & A_{N-1,N-1} \end{bmatrix},$$

such that each block $A_{i,j}$ is square and of size $n_b \times n_b$. If the matrix size n is not an integer multiple of n_b, then we embed A as the $n \times n$ leading principal sub-matrix of a slightly larger matrix B of size $\lceil n/n_b \rceil n_b \times \lceil n/n_b \rceil n_b$ and store B in the SBP format. We hence assume from now on that n is an integer multiple of n_b. In the SBP format, each block of A is stored contiguously in the column-major storage format and only the blocks in the lower triangular part of A are stored explicitly. For example, if $n = 8$ and $n_b = 2$, then using the SBP format we store the entries marked with a (below) as well as the redundant entries marked with $\star$:

$$A = \left[\begin{array}{cc|cc|cc|cc} a & \star & & & & & & \\ a & a & & & & & & \\ \hline a & a & a & \star & & & & \\ a & a & a & a & & & & \\ \hline a & a & a & a & a & \star & & \\ a & a & a & a & a & a & & \\ \hline a & a & a & a & a & a & a & \star \\ a & a & a & a & a & a & a & a \end{array}\right].$$

The blocks of A in the lower triangular part are stored contiguously in the following order:

$$A = \left[\begin{array}{c|c|c|c} 1 & & & \\ \hline 2 & 5 & & \\ \hline 3 & 6 & 8 & \\ \hline 4 & 7 & 9 & 10 \end{array}\right].$$

The motivation for storing also some redundant entries is that this makes the storage format simpler while only slightly increasing the storage requirements.

In Paper I [50], we define a tiled distributed-memory version of SBP and use it to implement a distributed-memory Cholesky factorization algorithm using packed storage.

3.5 QR Sweep

The classical QR algorithm (see, e.g., [43] and the references therein) reduces a real Hessenberg matrix H to real Schur form via an orthogonal similarity transformation. Specifically, the QR algorithm computes an orthogonal matrix Q such that $S = Q^T HQ$ is block upper triangular with square diagonal blocks of size 1×1 or 2×2, as illustrated in Figure 6. The eigenvalues of H can be read off from the main block diagonal of S: Each 2×2 block corresponds to a complex conjugate pair of eigenvalues and each 1×1 block corresponds to a real eigenvalue.

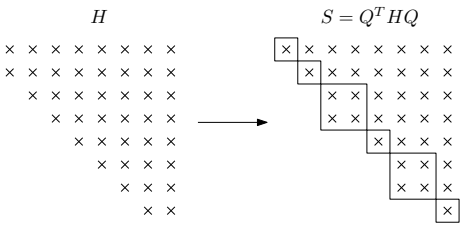


Figure 6: Illustration of a matrix H in Hessenberg form being reduced to a matrix S in real Schur form by an orthogonal similarity transformation.

One of the dominant computational kernels in the QR algorithm is the so called QR sweep, in which a bulge is introduced in the top left corner of the Hessenberg matrix and then chased down and eventually off the main diagonal by a sequence of orthogonal similarity transformations. The basic step is illustrated below for a 7×7 Hessenberg matrix H with an outlined 4×4 bulge on the main diagonal:

$$\underbrace{\begin{bmatrix} \times & \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times & \times \\ & h_1 & \times & \times & \times & \times & \times \\ & h_2 & \times & \times & \times & \times & \times \\ & h_3 & \times & \times & \times & \times & \times \\ & & & & \times & \times & \times \\ & & & & & \times & \times \end{bmatrix}}_{H} \mapsto \underbrace{\begin{bmatrix} \times & \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times & \times \\ & \alpha & \times & \times & \times & \times & \times \\ & 0 & \times & \times & \times & \times & \times \\ & 0 & \times & \times & \times & \times & \times \\ & & + & + & \times & \times & \times \\ & & & & & \times & \times \end{bmatrix}}_{Q^T HQ}.$$

By constructing a Householder reflection $\tilde{Q}$ of order 3 such that

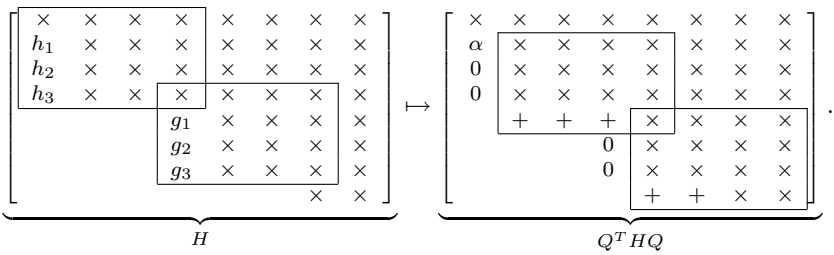
$$\tilde{Q}^T \begin{bmatrix} h_1 \\ h_2 \\ h_3 \end{bmatrix} = \begin{bmatrix} \alpha \\ 0 \\ 0 \end{bmatrix},$$

we can effectively move the bulge one step further down the main diagonal by subjecting H to the similarity transformation

$$H \leftarrow Q^T HQ \qquad \text{where} \quad Q = \begin{bmatrix} I_2 & & \\ & \tilde{Q} & \\ & & I_2 \end{bmatrix},$$

as illustrated above. The process can be repeated until the bulge disappears off of the bottom right corner of the matrix.

A drawback with bulge chasing is that a straightforward implementation of it has a very low arithmetic intensity if the matrix is too large to fit in cache. Part I [24] of the SIAG/LA 2003 Prize winning paper [25] explains how to chase not one but multiple small bulges as a tightly coupled chain, which enables more cache reuse and Level 3 performance. The example below illustrates the chasing of a chain consisting of two tightly coupled bulges one step down the main diagonal:



By constructing two Householder reflections $\tilde{Q}_1$ and $\tilde{Q}_2$ such that

$$\tilde{Q}_1^T \begin{bmatrix} h_1 \\ h_2 \\ h_3 \end{bmatrix} = \begin{bmatrix} \alpha \\ 0 \\ 0 \end{bmatrix} \qquad \text{and} \qquad \tilde{Q}_2^T \begin{bmatrix} g_1 \\ g_2 \\ g_3 \end{bmatrix} = \begin{bmatrix} \beta \\ 0 \\ 0 \end{bmatrix},$$

we can effectively move the two bulges by applying the transformation

$$H \leftarrow Q^T H Q \qquad \text{where} \quad Q = Q_1 Q_2 = Q_2 Q_1$$

and Q_1 and Q_2 are the natural extensions of $\tilde{Q}_1$ and $\tilde{Q}_2$ to the size of H, i.e.,

$$Q_1 = \begin{bmatrix} I_1 & & \\ & \tilde{Q}_1 & \\ & & I_4 \end{bmatrix} \qquad \text{and} \qquad Q_2 = \begin{bmatrix} I_4 & & \\ & \tilde{Q}_2 & \\ & & I_1 \end{bmatrix}.$$

By chasing a long chain of bulges several steps and delaying most of the updates, a comparatively large number of flops can be accumulated in a small orthogonal matrix Q and then applied efficiently using the Level 3 operation `GEMM` [25]. Figure 7 illustrates the general setting.

Figure 7 shows a chain of bulges being chased from the top left corner to the bottom right corner of the window block W by applying an orthogonal matrix Q of the same size as W to both sides of the matrix. The window block W is updated via the transformation

$$W \leftarrow Q^T W Q.$$

The two off-diagonal blocks R and L (see Figure 7) are updated via the transformations

$$R \leftarrow RQ \qquad \text{and} \qquad L \leftarrow Q^T L,$$

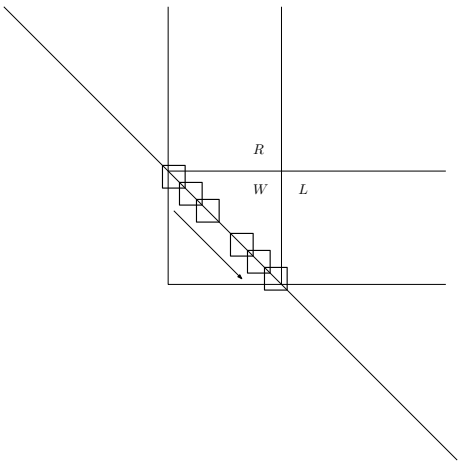


Figure 7: Illustration of the bulge chasing procedure in the small-bulge multi-shift QR algorithm [24]. The three bulges within the window block W are chased down the main diagonal via a specially constructed orthogonal matrix Q. The window block is updated via the transformation $W \leftarrow Q^T W Q$. The off-diagonal block R is updated from the right via $R \leftarrow RQ$, and the off-diagonal block L is updated from the left via $L \leftarrow Q^T L$.

thus completing a similarity transformation of the entire matrix.

In Paper IV [56], we develop and analyze a distributed-memory implementation of the updating part of the bulge-chasing procedure. We consider the problem of efficiently scheduling the operations on a block-cyclically distributed matrix in the case where the size of the window block W is about the same as the distribution block size. Simply performing (in parallel) first the update of R and then the update of L does not work very well since only a fraction of the processors ($\sqrt{p}$ or $2\sqrt{p}$ if p processors are arranged in a $\sqrt{p} \times \sqrt{p}$ mesh) are active in each of the updates.

Chapter 4

Summary of the Papers

Papers I–III [50, 55, 51] are primarily concerned with blocked matrix storage formats. In Paper I [50], we consider a distributed blocked and packed storage format and use it to implement a tiled distributed-memory implementation of right-looking Cholesky factorization with look-ahead. In Papers II–III [55, 51], we consider blocked storage formats for full matrices and, in particular, the efficient parallel and in-place conversion between all combinations of the four standard blocked storage formats and the standard row- and column-major formats.

Papers IV–VI [56, 57, 58] are concerned with the scheduling of two-sided transformations both for parallel efficiency and for cache reuse. In Paper IV [56], we investigate the scheduling of bulge chasing in a distributed-memory environment with the aim of improving the utilization of the processors. In Paper V [57], we improve the basic algorithm for the second stage of the two-stage Hessenberg reduction algorithm by reordering its operations to increase the arithmetic intensity. The problem of how to effectively schedule the operations on multi-core architectures is also considered in this paper. In Paper VI [58], we build on Paper V [57] and construct a novel parallel implementation of the full two-stage Hessenberg reduction algorithm. We also discuss the issues related to the implementation of the first stage and contribute a two-level blocked and parallel algorithm.

4.1 Paper I

We continue the work in [49] and present a distributed-memory realization of the Square Block Packed (SBP) format called DSBP. It allows the storage of a symmetric matrix using only slightly more than half the storage required for the full matrix. We also implement and evaluate three right-looking distributed-memory Cholesky factorization algorithms on top of the DSBP format. The Basic variant is comparable to the ScaLAPACK Cholesky factorization routine

PDPOTRF except that it uses the DSBP format and is a tiled algorithm. The Static variant improves on the Basic variant by using one level of look-ahead, which results in a near-optimal schedule in which the update of the next panel is interleaved with the factorization of that panel. Also, the tiles produced by the panel factorization are communicated to the other processors asynchronously and in an overlapped fashion. The Dynamic variant tries to further improve the Static variant by addressing a few issues observed in the schedules produced by the Static variant.

Experiments show that the idle time overhead in the Static variant is negligible on at least one of the processors. This suggests that no improvements to the schedules produced by the Static variant are possible without also distributing the work differently across the processors. Experiments show that the Dynamic variant performs almost identically to the Static variant, a result which can be explained by the observed near-optimality of the Static variant. The Basic variant as well as its closely related ScaLAPACK routine PDPOTRF perform quite similarly and significantly worse than both the Static and the Dynamic variants.

4.2 Paper II

We consider the problem of efficiently transposing a rectangular matrix in-place, i.e., changing the storage format of an $m \times n$ matrix A from column-major to row-major without using more than a constant amount of extra storage. We review a number of existing techniques for out-of-place, in-place, and semi in-place transposition including, for instance, the algorithms by Eklundh [40], Dow [39], and Alltop [4]. We propose a notation in which many of the transposition algorithms can be expressed and hence more easily be understood and compared.

The so called cycle-following algorithms are a key class of in-place transposition algorithms in which the cycle structure of the underlying permutation is exploited in order to shift the entries of the matrix around their corresponding cycles. The cycle-following algorithms have a minimal number of memory references, but they incur some overhead when finding the cycle structure. In this paper, we evaluate, among other things, the effectiveness of known algorithms for finding the cycle structure. In Paper III [51], we improve on this work by developing a new and more efficient algorithm for the problem of finding the cycle structure.

We show with experiments using sequential implementations that a three-stage, blocked, and in-place algorithm for matrix transposition compares favorably with both semi in-place and out-of-place algorithms, with the exception of Dow's V5 algorithm, which is the fastest of the considered algorithms for small matrices. For larger matrices however, the performance of the V5 algorithm deteriorates since the temporary buffer it uses grows too large for the cache and so the memory traffic increases substantially.

4.3 Paper III

This paper also considers the in-place transposition problem but focuses on the conversion of matrix storage formats in-place between the two standard formats (row- and column-major) and the four standard blocked formats (CCRB, CRRB, RCRB, and RRRB). We also develop parallel implementations targeting multi-core architectures. In-place transposition is the key computational kernel in the storage format conversion algorithms, and we base our work on cycle-following in-place transposition algorithms. At the core of these algorithms lies the problem of finding the cycle structure of the transposition permutation

$$P(k) = kn \bmod (mn - 1)$$

on the domain $0 < k < mn - 1$. Especially in a parallel environment, it is important to be able to find the cycle structure quickly and a priori in order to do effective load balancing across multiple processors/cores. Therefore, we develop in this paper a new algorithm, based on the ideas in [69], that is several orders of magnitude faster than previous algorithms (e.g., [85, 26]). We show via experiments on multi-socket and multi-core systems that our new algorithm reduces the overhead of finding the cycle structure down to negligible levels (e.g., fractions of 1%). Moreover, a one-stage conversion using our in-place algorithm is often faster than out-of-place conversion, besides using much less extra storage.

4.4 Paper IV

We consider the problem of improving a statically scheduled distributed-memory implementation of two-sided matrix computations by adding dynamic scheduling on the nodes and asynchronous and overlapped communications between the processors. As a model problem we consider the QR sweep. Traditional static scheduling techniques fail to deliver acceptable levels of efficiency and one has to resort to either multiple chains, alternative matrix distributions, or very large window sizes. The aim of this paper is to show that one can achieve high levels of efficiency even without these techniques. We show that the real problem with the QR sweep lies in the scheduling of the operations. The (node-)dynamic scheduler we propose builds an explicit DAG representation of the computations and communications on each node at run time based on an annotated (node-)sequential message-passing code. In order to control the schedule, we assign a priority to each task. We show via experiments that our priority-based dynamic scheduler can realize what we call a dual anti-diagonal wavefront pattern that effectively keeps the processors busy and raises the efficiency to more than 70% on 100 processors.

4.5 Paper V

This paper describes a new blocked and parallel algorithm for the critical part (Stage 2) of the two-stage Hessenberg reduction algorithm on multi-core architectures. The problem considered is that of reducing an $n \times n$ block Hessenberg matrix with $r \ll n$ non-zero sub-diagonals to Hessenberg form via an orthogonal similarity transformation. Known algorithms for symmetric band reduction are straightforward to generalize to the block Hessenberg case, but the standard implementation strategies, e.g., pipelining, do not yield acceptable levels of performance in the generalized case. The fundamental problem is that the accesses to the upper triangular part of the block Hessenberg matrix are difficult to organize in a manner that effectively uses the cache memories and the cores of a multi-core processor. In this paper, we contribute an efficient blocked and parallel algorithm that leverages techniques such as look-ahead and adaptive load balancing. Look-ahead hides the otherwise significant overhead caused by the equivalent of the panel factorizations in one-sided factorization algorithms. Adaptive load balancing yields coarse-grained tasks and a balanced load despite an inherently one-dimensional task decomposition and multiple barrier-style synchronization points per iteration of the main loop.

4.6 Paper VI

In this paper, we consider the complete two-stage Hessenberg reduction algorithm and compare it against the corresponding one-stage algorithms found in LAPACK and ScaLAPACK. We also describe a dynamically scheduled implementation of the first stage in detail. The task decomposition in the first stage is static and the two most critical operations are coarsely decomposed in order to maximize the performance of each task. Some of the updates are also delayed in order to reduce the idle time overhead otherwise caused by the panel factorizations. The look-ahead technique cannot be used in the first stage due to data dependencies.

Experiments show that our implementation of the two-stage algorithm is faster than the one-stage algorithm implemented in LAPACK and ScaLAPACK. The new code is faster both when only the Hessenberg matrix H as well as when both H and the orthogonal transformation matrix Q are computed explicitly.

Chapter 5

Other Publications

Besides the six papers included in this thesis, I have also been a co-author of the following two peer-reviewed papers (I and III) and my licentiate thesis (II).

I. Fred Gustavson, Lars Karlsson, and Bo Kågström. Three Algorithms for Cholesky Factorization on Distributed Memory using Packed Storage. In *Applied Parallel Computing: State of the Art in Scientific Computing (PARA 2006)*, LNCS 4699, pages 550-559, Springer, 2007.

II. Lars Karlsson. Blocked and Scalable Matrix Computations — Packed Cholesky, In-Place Transposition, and Two-Sided Transformations. Licentiate Thesis, Dept. of Computing Science, Umeå University, Sweden, 2009. Technical Report UMINF 09.11, ISBN 978-91-7264-788-6.

III. Bo Kågström, Lars Karlsson, and Daniel Kressner. Computing Codimensions and Generic Canonical Forms for Generalized Matrix Products. *Electronic Journal of Linear Algebra*, (to appear). Preprint available as Technical Report 2010-17, SAM, ETH Zurich, Switzerland.

Chapter 6

Conclusion and Future Work

The in-place storage format conversion algorithms are quite mature at this point, and they are currently used in the PLASMA library [2]. The low overhead incurred by our new algorithm for finding the cycle structure of the transposition permutation suggests that the approach should scale well to larger multi-core systems.

Since our early experiments with dynamic node-scheduling of two-sided transformations in Paper IV [56], the supercomputer architectures have changed considerably. The nodes now typically have multiple multi-core processors connected to some shared memory and possibly also some computational accelerators such as GPUs. Several large projects are currently developing dynamic schedulers for matrix computations aimed at multi-core systems with or without attached accelerators. Some attempts have also been made in the direction of distributed-memory systems. The issue of efficient message passing has not been thoroughly addressed though. In particular, there is a trade-off between overlap of communication with computation and latency that appears difficult to solve in a general way. It is likely that the MPI standard will need to be revised in order to better support dynamically scheduled algorithms. In particular, asynchronous notification of completed message transfers instead of polling or blocking is one desirable new feature.

The discovery that the second stage of two-stage Hessenberg reduction algorithm can be formulated as a blocked algorithm makes the two-stage reduction algorithm competitive and even faster than the standard one-stage algorithm. This opens up a great deal of possible future work. In the short term, the proposed algorithms need to be evaluated and adapted to emerging multi-core architectures. In the long term, the algorithms can be implemented and evaluated on distributed-memory systems. The algorithms might also generalize to two-stage Hessenberg-triangular reduction [35, 59], which opens up further possibilities. Another possible direction is to look at the use of tiled algorithms and/or blocked storage formats.

Bibliography

[1] E. Agullo, C. Augonnet, J. J. Dongarra, M. Faverge, J. Langou, H. Ltaief, and S. Tomov. LU Factorization for Accelerator-based Systems. Technical Report ICL-UT-10-05, ICL, University of Tennessee, December 2010.

[2] E. Agullo, J. Demmel, J. J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical Linear Algebra on Emerging Architectures: The PLASMA and MAGMA Projects. *Journal of Physics: Conference Series*, 180(1), 2009.

[3] E. Agullo, J. J. Dongarra, R. Nath, and S. Tomov. Fully Empirical Autotuned QR Factorization For Multicore Architectures. Technical Report arXiv:1102.5328, February 2011.

[4] W. O. Alltop. A Computer Algorithm for Transposing Nonsquare Matrices. *IEEE Transactions on Computers*, 24(10):1038–1040, 1975.

[5] W. Alvaro, J. Kurzak, and J. J. Dongarra. Fast and Small Short Vector SIMD Matrix Multiplication Kernels for the CELL Processor. Technical Report UT-CS-08-609, ICL, University of Tennessee, January 2008.

[6] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. J. Dongarra, J. Du Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D Sorensen. *LAPACK User's Guide (3rd ed.)*. Society for Industrial and Applied Mathematics, 1999.

[7] M. Anderson, G. Ballard, J. Demmel, and K. Keutzer. Communication-Avoiding QR Decomposition for GPUs. Technical Report UCB/EECS-2010-131, Electrical Engineering and Computer Sciences, University of California at Berkeley, 2010.

[8] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, December 2006.

[9] M. Bader. Exploiting the Locality Properties of Peano Curves for Parallel Matrix Multiplication. In *Proceedings of the 14th International Euro-Par Conf. on Parallel Processing*, pages 801–810. Springer-Verlag, 2008.

[10] M. Bader and C. Zenger. Cache Oblivious Matrix Multiplication Using an Element Ordering Based on a Peano Curve. *Linear Algebra and its Applications*, 417(2–3):301–313, 2006.

[11] R. M. Badia, J. R. Herrero, J. Labarta, J. M. Pérez, E. S. Quintana-Ortí, and G. Quintana-Ortí. Parallelizing Dense and Banded Linear Algebra Libraries using SMPSs. *Concurrency and Computation: Practice and Experience*, 21(18):2438–2456, 2009.

[12] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. Communication-Optimal Parallel and Sequential Cholesky Decomposition. *SIAM Journal on Scientific Computing*, 32(6):3495–3523, 2010.

[13] S. Barrachina, M. Castillo, F. D. Igual, R. Mayo, E. S. Quintana-Ortí, and G. Quintana-Ortí. Exploiting the Capabilities of Modern GPUs for Dense Matrix Computations. *Concurrency and Computation: Practice and Experience*, 21(18):2456–2477, 2009.

[14] P. Bientinesi, F. D. Igual, D. Kressner, M. Petschow, and E. S. Quintana-Ortí. Condensed Forms for the Symmetric Eigenvalue Problem on Multi-Threaded Architectures. *Concurrency and Computation: Practice and Experience*, (to appear).

[15] 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*, pages 340–347, Vienna, 1997.

[16] C. Bischof, B. Lang, and X. Sun. Parallel Tridiagonalization through Two-Step Band Reduction. In *Scalable High-Performance Computing Conference*, pages 23–27, 1994.

[17] C. H. Bischof, B. Lang, and X. Sun. A Framework for Symmetric Band Reduction. *ACM Transactions on Mathematical Software*, 26(4):581–601, 2000.

[18] C. H. Bischof and C. F. van Loan. The WY Representation for Products of Householder Matrices. *SIAM J. Sci. Statist. Comput.*, 8(1):S2–S13, 1987.

[19] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK User's Guide*. Society for Industrial and Applied Mathematics, 1997.

[20] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. An Updated Set of Basic Linear Algebra Subprograms (BLAS). *ACM Trans. Math. Software*, 28(2):135–151, 2002.

[21] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, September 1999.

[22] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, H. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, P. Luszczek, A. YarKhan, and J. J. Dongarra. Distributed-Memory Task Execution and Dependence Tracking within DAGuE and the DPLASMA Project. Technical Report ICL-UT-10-02, ICL, University of Tennessee, September 2010.

[23] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. J. Dongarra. DAGuE: A Generic Distributed DAG Engine for High Performance Computing. Technical Report ICL-UT-10-01, ICL, University of Tennessee, April 2010.

[24] 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.*, 23:929–947, 2001.

[25] K. Braman, R. Byers, and R. Mathias. The Multishift QR Algorithm. Part II: Aggressive Early Deflation. *SIAM J. Matrix Anal. Appl.*, 23:948–973, 2001.

[26] N. Brenner. Algorithm 467: Matrix Transposition in Place. *Communications of the ACM*, 16(11):692–694, 1973.

[27] A. Buttari, J. Langou, Kurzak J., and J. J. Dongarra. Parallel Tiled QR Factorization for Multicore Architectures. Technical Report UT-CS-07-598, ICL, University of Tennessee, 2007.

[28] A. Buttari, J. Langou, J. Kurzak, and J. J. Dongarra. A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures. *Parallel Computing*, 35(1):38–53, 2009.

[29] E. Chan. Runtime Data Flow Scheduling of Matrix Computations. Technical Report TR-09-22, The University of Texas at Austin, Department of Computer Sciences, August 2009.

[30] E. Chan, E. S. Quintana-Ortí, G. Quintana-Ortí, and R. van de Geijn. SuperMatrix Out-of-Order Scheduling of Matrix Operations for SMP and Multi-Core Architectures. In *SPAA '07: Proceedings of the 19th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 116–125, San Diego, CA, USA, June 2007.

[31] S. Chatterjee, V. V. Jain, A. R. Lebeck, S. Mundhra, and M. Thottethodi. Nonlinear Array Layouts for Hierarchical Memory Systems. In *Proceedings of the 13th International Conf. on Supercomputing*, pages 444–453. ACM, 1999.

[32] S. Chatterjee, A. R. Lebeck, P. K. Patnala, and M. Thottethodi. Recursive Array Layouts and Fast Parallel Matrix Multiplication. In *Proceedings of the Eleventh Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 222–231. ACM, 1999.

[33] J. Choi, J. J. Dongarra, S. Ostrouchov, A. P. Petitet, D. W. Walker, and R. C. Whaley. Design and Implementation of the ScaLAPACK LU, QR, and Cholesky Factorization Routines. *Scientific Programming*, 5(3):173–184, 1996.

[34] K. Dackland, E. Elmroth, B. Kågström, and C. F. Van Loan. Parallel Block Matrix Factorizations on the Shared Memory Multiprocessor IBM 3090 VF/600J. *International Journal of Supercomputer Applications*, 6(1):69–97, 1992.

[35] K. Dackland and B. Kågström. Blocked Algorithms and Software for Reduction of a Regular Matrix Pair to Generalized Schur Form. *ACM Transactions on Mathematical Software*, 25:425–454, 1999.

[36] J. Demmel and V. Volkov. LU, QR and Cholesky Factorizations using Vector Capabilities of GPUs. Technical Report UCB/EECS-2008-49, EECS Department, University of California, Berkeley, May 2008.

[37] J. W. Demmel, M. T. Heath, and H. A. van der Vorst. Parallel Numerical Linear Algebra. *Acta Numerica*, 2:111–197, 1993.

[38] J.J. Dongarra, S. Hammarling, and D.C. Sorensen. Block reduction of matrices to condensed forms for eigenvalue computations. *Journal of Computational and Applied Mathematics*, 27(1–2):215–227, 1989.

[39] M. Dow. Transposing a Matrix on a Vector Computer. *Parallel Computing*, 21(12):1997–2005, 1995.

[40] J. O. Eklundh. A fast computer method for matrix transposing. *IEEE Transactions on Computers*, 21(7):801–803, 1972.

[41] E. Elmroth, F. Gustavson, I. Jonsson, and B. Kågström. Recursive Blocked Algorithms and Hybrid Data Structures for Dense Matrix Library Software. *SIAM Review*, 46(1):3–45, 2004.

[42] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-Oblivious Algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, pages 285–298. IEEE Computer Society, 1999.

[43] G. H. Golub and C. F. van Loan. *Matrix Computations, Third Edition.* Johns Hopkins University Press, Baltimore, MD, USA, 1996.

[44] R. Granat, I. Jonsson, and B. Kågström. RECSY and SCASY Library Software: Recursive Blocked and Parallel Algorithms for Sylvester-Type Matrix Equations with Some Applications. In R. Ciegis et al., editor, *Parallel Scientific Computing–Advances and Applications*, volume 27, pages 3–24. Springer Optimization and Its Applications, 2009.

[45] R. Granat and B. Kågström. Algorithm 904: The SCASY Library—Parallel Solvers for Sylvester-Type Matrix Equations with Applications in Condition Estimation, Part II. *ACM Transactions on Mathematical Software*, 37(3):33:1–33:4, 2010.

[46] R. Granat and B. Kågström. Parallel Solvers for Sylvester-Type Matrix Equations with Applications in Condition Estimation, Part I: Theory and Applications. *ACM Transactions on Mathematical Software*, 37(3):32:1–32:32, 2010.

[47] F. G. Gustavson. Recursion Leads to Automatic Variable Blocking for Dense Linear Algebra Algorithms. *IBM Journal of Research and Development*, 41, 1997.

[48] 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, PARA'98*, Lecture Notes in Computer Science, LNCS 1541, pages 195–206. Springer, 1998.

[49] F. G. Gustavson, L. Karlsson, and B. Kågström. Three Algorithms for Cholesky Factorization on Distributed Memory Using Packed Storage. In *Applied Parallel Computing: State of the Art in Scientific Computing (PARA 2006)*, Lecture Notes in Computer Science, LNCS 4699, pages 550–559. Springer, 2007.

[50] F. G. Gustavson, L. Karlsson, and B. Kågström. Distributed SBP Cholesky Factorization Algorithms with Near-Optimal Scheduling. *ACM Trans. Math. Software*, 36(2):11:1–11:25, 2009.

[51] F. G. Gustavson, L. Karlsson, and B. Kågström. Parallel and Cache-Efficient In-Place Matrix Storage Format Conversion, (submitted).

[52] B. Hadri, H. Ltaief, E. Agullo, and J. J. Dongarra. Tall and Skinny QR Matrix Factorization Using Tile Algorithms on Multicore Architectures. Technical Report UT-CS-09-645, ICL, University of Tennessee, September 2009.

[53] A. Heinecke and M. Bader. Towards Many-Core Implementation of LU Decomposition using Peano Curves. In *On Computing Frontiers, UCHPC-MAW '09: Proceedings of the combined workshops on UnConventional high*

performance computing workshop plus memory access workshop, pages 21–30. ACM, May 2009.

[54] T. Joffrain, T. M. Low, E. S. Quintana-Ortí, R. van de Geijn, and F. Van Zee. Accumulating Householder Transformations, Revisited. *ACM Transactions on Mathematical Software*, 32(2):169–179, 2006.

[55] L. Karlsson. Blocked In-Place Transposition with Application to Storage Format Conversion. Technical Report UMINF 09.01, Department of Computing Science, Umeå University, SE-901 87 Umeå, Sweden, January 2009.

[56] L. Karlsson and B. Kågström. A Framework for Dynamic Node-Scheduling of Two-Sided Blocked Matrix Computations. In *Proceedings of PARA 2008*, LNCS. Springer, (to appear).

[57] L. Karlsson and B. Kågström. Efficient Reduction from Block Hessenberg Form to Hessenberg Form using Shared Memory. In *Proceedings of PARA 2010*, LNCS. Springer, (to appear).

[58] L. Karlsson and B. Kågström. Parallel Two-Stage Reduction to Hessenberg Form using Dynamic Scheduling on Shared-Memory Architectures, (to appear).

[59] B. Kågström, D. Kressner, E. S. Quintana-Ortí, and G. Quintana-Ortí. Blocked Algorithms for the Reduction to Hessenberg-Triangular Form Revisited. *BIT Numerical Mathematics*, 48(3):563–584, 2008.

[60] B. Kågström, P. Ling, and C. Van Loan. Algorithm 784: GEMM-Based Level 3 BLAS: Portability and Optimization Issues. *ACM Trans. Math. Software*, 24(3):303–316, 1998.

[61] 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*, 24:268–302, 1998.

[62] J. Kurzak and J. J. Dongarra. Parallel Block Hessenberg Reduction using Algorithms-By-Tiles for Multicore Architectures Revisited. Technical Report UT-CS-08-624, ICL, University of Tennessee, August 2008.

[63] J. Kurzak and J. J. Dongarra. QR Factorization for the Cell Broadband Engine. *Scientific Programming*, 17(1–2):31–42, 2009.

[64] J. Kurzak, H. Ltaief, J. J. Dongarra, and R. M. Badia. Scheduling Dense Linear Algebra Operations on Multicore Processors. *Concurrency and Computation: Practice and Experience*, 22(1):15–44, 2010.

[65] M. D. Lam, E. E. Rothberg, and M. E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. *SIGARCH Computer Architecture News*, 19(2):63–74, 1991.

[66] H. Ltaief, J. Kurzak, and J. J. Dongarra. Parallel Two-Sided Matrix Reduction to Band Bidiagonal Form on Multicore Architectures. *IEEE Transactions on Parallel and Distributed Systems*, 21(4):417–423, 2010.

[67] H. Ltaief, J. Kurzak, J. J. Dongarra, and R. M. Badia. Scheduling Two-sided Transformations using Tile Algorithms on Multicore Architectures. *Scientific Programming*, 18(1):35–50, 2010.

[68] K. Murata and K. Horikoshi. A New Method for the Tridiagonalization of the Symmetric Band Matrix. *Information Processing in Japan*, 15:108–112, 1975.

[69] G. Pall and E. Seiden. A Problem in Abelian Groups, with Application to the Transposition of a Matrix on an Electronic Computer. *Mathematics of Computation*, 14(70):189–192, 1960.

[70] N. Park, B. Hong, and V. K. Prasanna. Tiling, Block Data Layout, and Memory Hierarchy Performance. *IEEE Transactions on Parallel and Distributed Systems*, 14(7):640–654, 2003.

[71] J. M. Perez, R. M. Badia, and J. Labarta. A Flexible and Portable Programming Model for SMP and Multi-cores. Technical Report 03/2007, Barcelona Supercomputing Center, 2007.

[72] M. Petschow and P. Bientinesi. MR3-SMP: A Symmetric Tridiagonal Eigensolver for Multi-Core Architectures. Technical Report AICES-2010/10-2, Aachen Institute for Advanced Study in Computational Engineering Science, October 2010.

[73] J. Poulson, B. Marker, and R. van de Geijn. Elemental: A New Framework for Distributed Memory Dense Matrix Computations. Technical Report TR-10-20, The University of Texas at Austin, Department of Computer Sciences, June 2010.

[74] G. Quintana-Ortí, E. S. Quintana-Ortí, R. van de Geijn, F. G. Van Zee, and E. Chan. Programming Matrix Algorithms-by-Blocks for Thread-Level Parallelism. *ACM Transactions on Mathematical Software*, 36(3):14:1–14:26, 2009.

[75] G. Quintana-Ortí and R. A. van de Geijn. Improving the Performance of Reduction to Hessenberg Form. *ACM Trans. Math. Software*, 32(2):180–194, 2006.

[76] G. Quintana-Ortí, F. D. Igual, E. S. Quintana-Ortí, and R. A. van de Geijn. Solving Dense Linear Systems on Platforms with Multiple Hardware Accelerators. *SIGPLAN Notices*, 44:121–130, February 2009.

[77] H. Rutishauser. On Jacobi rotation patterns. In *Proc. Sympos. Appl. Math.*, volume XV, pages 219–239, Providence, R.I., 1963. Amer. Math. Soc.

[78] R. Schreiber and C. F. van Loan. A Storage-Efficient WY Representation for Products of Householder Transformations. *SIAM J. Sci. Statist. Comput.*, 10(1):53–57, 1989.

[79] H. R. Schwarz. Tridiagonalization of a symmetric band matrix. *Numerische Mathematik*, 12(4):231–241, 1968.

[80] F. Song, A. YarKhan, and J. Dongarra. Dynamic Task Scheduling for Linear Algebra Algorithms on Distributed-Memory Multicore Systems. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 19:1–19:11, New York, NY, USA, 2009. ACM.

[81] S. Tomov and J. J. Dongarra. Accelerating the Reduction to Upper Hessenberg Form Through Hybrid GPU-Based Computing. Technical Report UT-CS-09-642, University of Tennessee Computer Science, May 2009. Also as LAPACK Working Note 219.

[82] C. F. Van Loan. Using the Hessenberg decomposition in control theory. In *Algorithms and Theory in Filtering and Control*, Mathematical Programming Studies, pages 102–111. North-Holland, 1982.

[83] F. G. Van Zee, E. Chan, R. van de Geijn, E. S. Quintana-Ortí, and G. Quintana-Ortí. Introducing the libflame Library for Dense Matrix Computations. *IEEE Computing in Science and Engineering*, 11(6):56–62, 2009.

[84] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated Empirical Optimization of Software and the ATLAS Project. *Parallel Computing*, 27(1–2):3–35, 2001.

[85] P. F. Windley. Transposing Matrices in a Digital Computer. *The Computer Journal*, 2(1):47–48, 1959.