



Moritz Klammler

Application of Efficient Matrix Inversion to the Decomposition of Hierarchical Matrices

Bachelor's Thesis

Date of Submission: November 11, 2014

Supervisor: Prof. Dr. Peter Sanders
Dipl.-Inform., Dipl.-Math. Jochen Speck

Secondary Supervisor: Prof. Dr. Christian Wieners
Dr. Daniel Weiß

Institute of Theoretical Informatics, Algorithmics
Department of Informatics

Institute for Applied and Numerical Mathematics
Department of Mathematics

Karlsruhe Institute of Technology

Copyright © 2014 Moritz Klammler

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

The Software is provided “as is”, without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose and noninfringement. In no event shall the authors or copyright holders be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the Software or the use or other dealings in the Software.

Contents

1	Introduction	1
1.1	Previous Work	1
1.2	Our Contribution	2
1.3	Overview	3
2	Preliminaries	4
2.1	Typographical Conventions	4
2.2	Complexity	5
2.2.1	Asymptotic Complexity	7
2.2.2	Floating Point Operations	8
2.2.3	Execution Time	9
2.3	Linear Systems	11
2.3.1	<i>LU</i> Decomposition	11
2.3.1.1	Forward Substitution	12
2.3.1.2	Backward Substitution	13
2.3.2	Cholesky Decomposition	14
2.3.2.1	Inner Product Cholesky	15
2.3.2.2	Gaxpy Cholesky	17
2.3.2.3	Outer Product Cholesky	19
2.3.3	Block <i>LU</i> Decomposition	19
2.3.4	Block <i>LDU</i> Decomposition	20
2.3.5	Matrix Inversion	20
2.3.5.1	Gauß-Jordan Elimination	21
2.3.5.2	Strassen Inversion	21
2.3.5.3	Newton Inversion	23
2.3.5.4	The NeSt Algorithm	25
2.4	Hierarchical Matrices	27
2.4.1	Nested Dissection	28
2.4.1.1	Symmetry	30
3	Algorithmics	32
3.1	Notation	33
3.2	Block <i>LL</i> ^T Decomposition of H-Matrices	34

3.2.1	Decomposition	34
3.2.2	Work-Flow	36
3.2.3	Complexity	37
3.2.4	Parallelism	40
3.2.4.1	Synchronization	40
3.2.4.2	Parallel Efficiency	42
3.2.5	Solving	42
3.3	Block LDL^T Decomposition of H-Matrices	43
3.3.1	Complexity	46
3.3.2	Parallelism	46
4	Implementation	48
4.1	Technologies	48
4.1.1	Programming Languages	48
4.1.2	Libraries	49
4.1.2.1	BLAS & LAPACK	49
4.1.2.2	Boost uBLAS	50
4.1.2.3	TNT	52
4.1.2.4	Eigen	53
4.1.2.5	Comparison	55
4.1.3	Tools and Programs	59
4.2	Algorithms and Data Structures	59
4.2.1	Data Structures for Hierarchical Matrices	60
4.2.1.1	The Abstract <code>HMatrix</code> Class	60
4.2.1.1.1	Visitors	61
4.2.1.2	Leafs: The <code>FullMatrix</code> Class	62
4.2.1.3	Nested Dissection: The <code>BlockMatrix</code> Class	62
4.2.1.4	Arbitrary Children: The <code>ArrangedMatrix</code> Class	62
4.2.1.5	Slicing and Transposing: The <code>MatrixView</code> Class	63
4.2.1.6	HVector and FullVector	65
4.2.2	Basic Algorithms for Hierarchical Matrices	65
4.2.2.1	Implementation Strategy	65
4.2.2.2	Sums and Differences	67
4.2.2.3	Products	69
4.2.2.3.1	Matrix-Vector Products	69
4.2.2.3.2	Matrix-Matrix Products	70
4.2.3	Decompositions	72
4.3	Parallelization	73
4.3.1	Synchronization	75
4.4	Observation	77
4.5	Quality	78

5	Experimental	80
5.1	Test Setup	81
5.2	Hardware	81
5.3	Decomposition	82
5.4	Solving	84
5.5	Accuracy	84
5.6	Tests on the Smaller ITI-120	89
6	Conclusion	91
6.1	Further Work	92
	Bibliography	I
	List of Algorithms	III
	List of Code Listings	V
	List of Figures	VII
	List of Tables	IX

Acknowledgments

I want to thank my parents for their unconditional life-long support during all the years of my academic career.

Furthermore, I would like to thank Daniel Weiß for giving a great lecture on numerical mathematics that has attracted me to the subject. I would like to thank my supervisors Jochen Speck and Daniel Weiß for always having been there for me during preparation of this work. I'm especially thankful to Daniel Maurer for many hours of inspiring discussions about the subject. Without his input, this work would not have been possible.

Last but not least, I would like to thank the hackers from the Eigen project for having built such a great linear algebra library and releasing it as free software. Eigen's clean interfaces and outstanding performance have made implementing this work much more joyful. In particular, I am thankful to Christoph Hertzberg from the Eigen project for promptly replying to our inquiries.

Moritz Klammmler, November 2014

Chapter 1

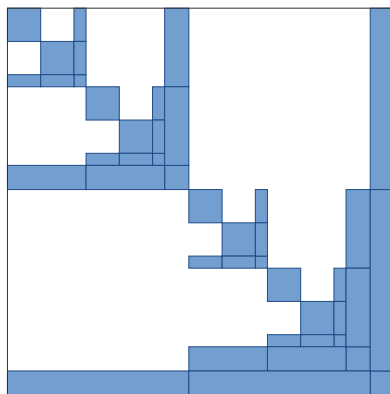
Introduction

Solving linear systems – that is, given $\mathbf{M} \in \mathbb{R}^{n \times n}$ and $\mathbf{b}^{(1)}, \dots, \mathbf{b}^{(k)} \in \mathbb{R}^n$ for $n, k \in \mathbb{N}$, finding $\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(k)} \in \mathbb{R}^n$ such that

$$\mathbf{M}\mathbf{x}^{(l)} = \mathbf{b}^{(l)}$$

for $l \in \{1, \dots, k\}$ – is one of the fundamental tasks of numerical mathematics (§ 2.3).

We are studying systems where the coefficient matrix \mathbf{M} is symmetric and positive definite and has a special hierarchical structure that is obtained by performing a procedure known as *nested dissection* (§ 2.4.1) that yields a structure like the following



where the blank regions are known to be zero and the shaded blocks may contain anything. Such *hierarchical matrices* (*H-matrices*) are stored as a tree of submatrices with ordinary matrix representations at the leaves. Many operations for such matrices can be formulated elegantly using recursion.

1.1 Previous Work

Maurer and Wieners have adapted block *LU* decomposition (§ 2.3.4) to such matrices and presented an implementation closely related to finite-element compu-

tations for shared memory systems [15].

Meanwhile, Sanders, Speck, and Steffen have presented an algorithm that combines Newton (§ 2.3.5.3) and Strassen (§ 2.3.5.2) inversion to obtain a work-efficient matrix inversion algorithm (NeSt, § 2.3.5.4) for symmetric and positive definite matrices with poly-logarithmic time complexity [20].

1.2 Our Contribution

We have combined block LU decomposition of H-matrices with nested dissection structure – which is block LL^T decomposition for symmetric and positive definite matrices – (§ 3.2) with matrix inversion to obtain a new scheme that we call block LDL^T decomposition (§ 3.3). It computes a decomposition into a lower triangular block matrix and a block diagonal matrix where the latter is inverted.

Using NeSt for matrix inversion, this algorithm’s work is dominated almost exclusively by computing matrix-matrix products for which the most highly optimized libraries are available off-the-shelf. Furthermore, it is important that matrix-matrix multiplication has a logarithmic critical path length if computed in parallel while Cholesky factorization (§ 2.3.2) and substitution (§ 2.3.1.1) which are needed for the block LL^T version both have a linear critical path. Therefore, our variant is supposed to take out more of the hardware and scale better to highly parallel systems. On the down side, the LDL^T variant has to perform more work for the decomposition.

Once the decomposition is computed, both algorithms can solve systems for the various right-hand sides with equal work required. However, the LDL^T variant has again a shorter critical path and only needs matrix-vector products, which are again readily available in highly optimized versions.

We have implemented block LL^T and LDL^T decomposition for shared-memory systems using the C++ programming language (§ 4). The source code is available from the author as free software ¹.

Tests (§ 5) show that for a machine with 32 CPUs and 500 GiB memory, the achieved ratio of effective floating point instructions per unit time is better roughly by a factor of two which sometimes compensates for the additional work that is to be done so with regard to overall execution time, there are problems where either of the algorithms outperforms the other. However, the advantage of block LDL^T is not significant and for smaller systems, the LL^T version is much faster.

On the other hand, we could show a significant improvement on the execution time when it comes to solving individual linear systems, once a decomposition is computed.

Other than one might expect, the additional errors introduced by computing the inverse matrix via an iterative procedure – as opposed to a direct Cholesky factorization – are significant but very moderate ($\approx 10\%$), at least for the well-conditioned systems we have investigated.

¹<http://www.klammler.eu/bsc/>

1.3 Overview

In chapter 2 we introduce some fundamental concepts that are required for our work. Most of this is textbook knowledge and may be skipped by the enlightened reader. Section 2.3 depends on section 2.2 but section 2.4 is mostly independent. The reader acquainted with the concept of H-matrices might be interested to hear that we do not consider low-rank representations [4] in our work but assume all non-zero blocks be stored as full matrices. The sub-sections describing the various decompositions in section 2.3 are again mostly independent of each other. All of this writing depends on the typographical conventions introduced in section 2.1.

Chapter 3 presents the block LL^T and block LDL^T decomposition of symmetric and positive definite H-matrices with nested dissection structure. The algorithms are discussed on an abstract level.

Then, in chapter 4 we describe our specific implementation. The section 4.1 can be read in isolation but section 4.2 depends on the previous chapter. The remaining sections in this chapter depend on section 4.2. Some knowledge of the C++ programming language is assumed for the whole chapter.

In chapter 5 we present some experimental results from tests with running our implementation on two server machines. It requires a brief understanding of what we did but might be digestible in isolation for a reader used to the subject after reading only the present chapter.

Finally, we draw a conclusion in chapter 6.

Chapter 2

Preliminaries

2.1 Typographical Conventions

If all possible, we use upper-case Latin symbols ($\mathbf{A}, \mathbf{B}, \mathbf{C}, \dots$) for matrices and lower-case Latin symbols ($\mathbf{u}, \mathbf{v}, \mathbf{w}, \dots$) for vectors. Symbols denoting matrices or vectors are printed in bold. We write $\mathbf{1}$ and $\mathbf{0}$ for the unit matrix and a matrix of all zeros respectively where the dimensions will be unambiguous from the context and likewise for vectors. For real scalar factors, we use lower-case Greek letters ($\alpha, \beta, \gamma, \dots$). The lower-case Latin symbols i, j, k, l, m, n, \dots are used for integral values (mostly dimensions and indices). Variables referring to commonly used domains (integers, reals, ...) are printed in blackboard bold ($\mathbb{N}, \mathbb{R}, \dots$). We use the symbol \mathbb{N} for the positive, \mathbb{N}_0 for the non-negative and \mathbb{Z} for the set of all integers. Unless otherwise mentioned, matrices, vectors and scalars are real.

Here are some examples:

$$\mathbf{Ax} = \mathbf{b}$$

This is the most simple form of a matrix equation. \mathbf{A} is a matrix and \mathbf{x} and \mathbf{b} are vectors.

$$(\lambda \mathbf{1} - \mathbf{A})\mathbf{v} = \mathbf{0}$$

This is the canonical definition of the eigenvalue problem. λ is a scalar, $\mathbf{1}$ the unit matrix, \mathbf{A} a matrix, \mathbf{v} a vector and $\mathbf{0}$ the zero vector.

Indexing In equations and pseudo-code, we are using 1-based indexing unless mentioned otherwise. For example,

$$v_i = \sum_{k=1}^m A_{ik} u_k$$

is the element-wise definition of the matrix-vector product $\mathbf{Au} = \mathbf{v}$ with $\mathbf{v} \in K^n$, $\mathbf{u} \in K^m$ and $\mathbf{A} \in K^{n \times m}$ for field K and $n, m \in \mathbb{N}$. The variables are not bolded

because the *entries* in a matrix are scalars (elements in $A_{ij} \in K$; of course, there are also non-scalar fields, but we don't consider them).

Sub-Matrices & Slicing Syntax If needed, we use the slicing syntax “[$i : j$]” found in some high-level programming languages to refer to sub-matrices and sub-vectors.

For a vector \mathbf{v} , the expression $\mathbf{v}[i : j]$ refers to the sub-vector with the elements v_i, \dots, v_j (all inclusive) and is only meaningful if $1 \leq i \leq j \leq n$ where n is the size of \mathbf{v} . The expression $\mathbf{v}[:]$ is the same as \mathbf{v} but this verbose syntax is needed to disambiguate matrix slices. For a matrix \mathbf{A} , the expression $\mathbf{A}[i][:]$ extracts the i -th row vector of \mathbf{A} while $\mathbf{A}[:,j]$ extracts the j -th column vector. The expression $\mathbf{A}[:,:]$ is identical to \mathbf{A} and $\mathbf{A}[i][j]$ is a 1×1 matrix with the only element A_{ij} .

We write $\mathbf{B} \sqsubseteq \mathbf{A}$ to express that the matrix \mathbf{B} is a sub-matrix of \mathbf{A} . That is, if $\mathbf{A} \in K^{n_A \times m_A}$ and $\mathbf{B} \in K^{n_B \times m_B}$ for some field K and $n_A, m_A, n_B, m_B \in \mathbb{N}$, then the expression $\mathbf{B} \sqsubseteq \mathbf{A}$ means that there exist $i_l, i_u \in \{1, \dots, n_A\}$ and $j_l, j_u \in \{1, \dots, m_A\}$ such that $\mathbf{B} = \mathbf{A}[i_l : i_u][j_l : j_u]$.

Bra-Ket Notation We don't treat row and column vectors differently, that is, we use the vectors \mathbf{v} and \mathbf{v}^T interchangeably. It will be clear from the context how the vector is to be applied. Where this is not clear, we resort to the bra-ket syntax to disambiguate. If \mathbf{u} and \mathbf{v} are vectors, then $\langle \mathbf{u} |$ is a row vector and $|\mathbf{v}\rangle$ is a column vector. Therefore, $\langle \mathbf{u} | \mathbf{v} \rangle$ is an inner (dot) product and $|\mathbf{u}\rangle \langle \mathbf{v} |$ is an outer (tensor) product.

Inverse and Transpose Since for any regular square matrix \mathbf{A} , the transpose of the inverse $(\mathbf{A}^{-1})^T$ and the inverse of the transpose $(\mathbf{A}^T)^{-1}$ are identical, we use the shorthand notation \mathbf{A}^{-T} to refer to either of them.

2.2 Complexity

The complexity of a procedure describes how the consumption of a precious resource (execution time, memory, computation units, ...) scales with the size of the input. Such resource consumption is frequently referred to as *cost*.

DEFINITION 1 (COST FUNCTION) Let \mathcal{R} be a resource that can be measured (or approximated) by an integral quantity. Let further Σ be an alphabet (without loss of generality, assume $\Sigma = \{0, 1\}$) and $p : \Sigma^* \rightarrow \Sigma^*$ a Turing-commutable function. Finally, let \mathcal{A} be an algorithm that computes p and $r : \mathbb{N}_0 \rightarrow \mathbb{N}_0$. r is a **cost function** for \mathcal{A} 's consumption of resource \mathcal{R} if $r(|w|)$ is the (peak) consumption of resource \mathcal{R} by \mathcal{A} while computing $p(w)$.

Naturally, one has an interest in keeping the growth of the cost function, as well as its magnitude as low as possible. For a given problem, if there is more

than one known algorithm to solve it, the best of these algorithms defines an upper bound to the theoretical complexity of the problem as a whole. For some problems, theoretical proofs exist, that the bound is sharp, ie there can be no better algorithm.

While definition 1 of a cost function is (as) general (as the Turing computation model), it is often useful to use a somewhat modified definition of input size that is closer to the problem domain.

As an example, consider an algorithm that operates on a real matrix $A \in \mathbb{R}^{n \times n}$ with $n \in \mathbb{N}$.

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}$$

Since real numbers cannot be represented in general by a finite sequence of digits, we'll approximate the matrix entries a_{ij} by fixed-size keys $\hat{a}_{ij} \in \{1, \dots, 2^d\}$ with constant $d \in \mathbb{N}$ into a finite lookup-table of some carefully chosen real numbers that hopefully minimize rounding errors.¹

$$\hat{A} = \begin{pmatrix} \hat{a}_{11} & \hat{a}_{12} & \cdots & \hat{a}_{1n} \\ \hat{a}_{21} & \hat{a}_{22} & \cdots & \hat{a}_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ \hat{a}_{n1} & \hat{a}_{n2} & \cdots & \hat{a}_{nn} \end{pmatrix} \approx A$$

Using the alphabet $\Sigma = \{0, 1, [,], , , ;\}$, we can now encode the approximated input \hat{A} as a string $s = [s_{11} , s_{12} , \dots , s_{1n} ; s_{21} , s_{22} , \dots , s_{2n} ; \dots ; s_{n1} , s_{n2} , \dots , s_{nn}]$ where the $s_{ij} \in \{0, 1\}^d$ are the binary encoded keys \hat{a}_{ij} .

It can be readily seen that the length of this encoding is given by $|s| = (1 + d)n^2 + 1$. However, that's pretty far from the problem domain (real linear algebra) and depends on unimportant (and arbitrarily chosen) details such as the exact syntax² of the encoding and the bit size d of the real value type. For most discussions, it would be much more convenient to measure complexity directly as a function of n , where the reader will understand that there is a simple mapping of n to the length of a reasonable encoding. We will use this adapted (loose) definition of cost functions for the remainder of this work but include some remarks on encoding where appropriate. It is worth mentioning that the above approach to encoding is using the Turing computation model whereas all modern hardware uses random access memory. We will discuss complexity in terms of real hardware, not hypothetical Turing machines.

¹This is just an awkward way to think of fixed-size floating point numbers but this theoretical approach allows us to discuss lossy encodings of real numbers without taking a stand for any particular convention such as 32 bit IEEE 754 floating point numbers.

²Which – in this example – “happens” to be the one understood by the user interfaces of some popular linear algebra software packages.

2.2.1 Asymptotic Complexity

In theoretical computer science, it is often interesting how the cost of an algorithm scales if the input size approaches infinity. Biases and constant factors are generally less interesting because they are readily overcome by the steadily increasing power of computers.

DEFINITION 2 (ASYMPTOTIC GROWTH) Let $f : \mathbb{N}_0 \rightarrow \mathbb{R}$ be a function (or a sequence, if you prefer).

The set

$$O(f) = \{(g : \mathbb{N}_0 \rightarrow \mathbb{R}) : \exists \alpha > 0, N \in \mathbb{N}_0 : \forall n \geq N : |f(n)| \geq \alpha |g(n)|\} \quad (2.1)$$

is defined to contain all functions g that grow *at most as fast* as f . We say that all of the g are **asymptotically bounded above** by f , ignoring constant factors.

Likewise, the set

$$\Omega(f) = \{(g : \mathbb{N}_0 \rightarrow \mathbb{R}) : \exists \alpha > 0, N \in \mathbb{N}_0 : \forall n \geq N : |f(n)| \leq \alpha |g(n)|\} \quad (2.2)$$

is defined to contain all functions g that grow *at least as fast* as f . We say that all of the g are **asymptotically bounded below** by f , ignoring constant factors.

Finally, the set

$$\Theta(f) = O(f) \cap \Omega(f) \quad (2.3)$$

is defined as the intersection of the above two. It contains all functions g that grow *as fast* as f . We say that all of the g are **asymptotically bounded above and below** by f , again ignoring constant factors.

Using the symbols from definition 2 is also known as *Landau notation*³.

To simplify the notation, we will often write for example $O(n^3)$ which will be understood as $O(n \mapsto n^3)$.

When using asymptotic growth to discuss the complexity of cost functions, care has to be taken that this ignores (on purpose) any constant factors. For example, algorithm \mathcal{A} and \mathcal{B} might both compute the same function. The peak amount of memory required to process an input of n bytes size might be given by αn^2 bytes for \mathcal{A} and βn^3 bytes for \mathcal{B} . Clearly, for $n > \alpha/\beta$, algorithm \mathcal{A} will use less memory. However, if α/β is so large, that current hardware cannot realistically handle it, algorithm \mathcal{B} will perform better for all inputs of feasible size while algorithm \mathcal{A} will have to be abandoned until the hardware of the future will eventually allow for computing large enough problems.

³In honor of the German mathematician Edmund Georg Hermann Landau (* 1877, Berlin, Germany; † 1938, ebenda).

2.2.2 Floating Point Operations

To reason about the amount of work an algorithm has to carry out, we like to look at the number of floating point operations it has to perform. Since at the lowest level, most computations basically consist of computing sums of products, the following definition appears useful.

DEFINITION 3 (FLOATING POINT OPERATION) An **effective floating point operation (FLOP)** is a pair of a multiplication (or division) and addition (or subtraction) of floating point values that is *required* by an algorithm in order to compute its result.

When “counting” the FLOPs in an algorithm, we are generally only interested in the term that dominates the asymptotic growth. However, we generally do not use Landau-notation because constant factors usually do matter.

For example, if the exact number of FLOPs for input size $n \in \mathbb{N}$ were given by

$$W(n) = c_0 + c_1n + c_2n^2$$

we will be fine with knowing that for large n , the amount of work is basically given by $W(n) \approx c_2n^2$. However, we will not rush to say that $W(n) \in \Theta(n^2)$, although it is correct, because it does matter in practice how large the constant factor c_2 is.

FLOPs are counted on an abstract algorithmic level without worrying about implementation specific details. For example, the scalar product of the vectors \mathbf{u} and \mathbf{v}

$$\langle \mathbf{u} | \mathbf{v} \rangle = \sum_{i=1}^n u_i v_i \tag{2.4}$$

with $\mathbf{u}, \mathbf{v} \in \mathbb{R}^n$ and $n \in \mathbb{N}$ takes n FLOPs, when computed directly as in algorithm 2.1.

Even ignoring the machine instructions needed for branching, at the very least, the computer will have to do n additions of integers to keep track of the iteration index i . We deliberately did not include them in our estimate, not so much because it is an integer and not a floating point operation (which are often faster) but because it is not something that is actually useful from a numeric point of view. In fact, a concerned programmer or (better) a smart compiler could easily reduce the number of i -increments needed by any constant factor by trivially unrolling the loop.

If above algorithm were implemented in some higher-level language, it might very well take even more machine instructions just to load a value from an array (eg for bounds checking or dynamic binding). Would we have to take all these operations into account, it would no longer be possible to reason about the general idea of an algorithm as an abstract concept. Instead, we would be discussing implementation details of compilers and hardware that might quickly become obsolete.

PROCEDURE DotProduct**INPUT** $\mathbf{u}[1 \dots n] : \text{REAL}$ $\mathbf{v}[1 \dots n] : \text{REAL}$ **OUTPUT** $d : \text{REAL}$ **VARIABLES** $i : \text{INTEGER}$ **BEGIN** $d \leftarrow 0$;; *constant term ignored***FOR** $i \leftarrow 1$ **TO** n **DO** ;; *loop control overhead ignored* $d \leftarrow d + u_i v_i$;; *1 FLOP per iteration***DONE****END**

ALGORITHM 2.1: Example algorithm computing the inner (“dot”) product of two real vectors $\mathbf{u}, \mathbf{v} \in \mathbb{R}^n$ as in equation 2.4. This algorithm performs n FLOPs in total. The single instruction for initializing d with 0 is ignored as is the implementation specific overhead for loop control and the call-return overhead.

By only counting effective FLOPs, we can give a measure of the theoretical cost of an algorithm that will be applicable to any implementation.

2.2.3 Execution Time

From a user’s perspective, what is even more interesting than the number of operations an algorithm carries out, is how long they will have to wait until it eventually completes computation.

DEFINITION 4 (EXECUTION TIME) The **execution time** of a program is the amount of physical time that passes after the program is started until it halts.

We say “physical time” rather than “wall time” or “world time” because we are referring to the physical quantity time, not the man-made convention of dates and times that is crippled by leap seconds and clock adjustments.

Measuring physical time – as opposed to counting machine instructions – is agnostic with regard to any hardware or software details. It will therefore automatically take into account any overhead that is created by cache misses, pipeline flushes, system calls, thread synchronization and even effects we might not even be aware of.

To give a measure how well an implementation makes use of the hardware, we compare the fraction of effective FLOPs per unit execution time.

DEFINITION 5 (FLOP RATE) Let \mathcal{A} be a numeric algorithm where the number

of FLOPs for a problem of given size is given by the cost function $W : \mathbb{N}_0 \rightarrow \mathbb{N}_0$. Let further \mathcal{I} be an implementation of \mathcal{A} where the execution time for an input of given size on a fixed number $c \in \mathbb{N}$ of cores is given by $t_c : \mathbb{N}_0 \rightarrow \mathbb{R}$. The **FLOP rate**

$$R_c(n) = \frac{W(n)}{ct_c(n)} \quad (2.5)$$

is a quantity that denotes the number of effective FLOPs that – on average – are carried out on each core per unit time.

In SI units, the unit of R would be FLOPs per second and core, conveniently abbreviated to FPS per core.

A high FLOP rate indicates good usage of the hardware and low bookkeeping overhead. In no way does the FLOP rate have to or will be constant for varying inputs or numbers of cores.

FLOP count and FLOP rate are orthogonal. While the former measures how much (or, preferably, how little) numeric work needs to be done by an algorithm, the latter measures how effective an algorithm is implemented by a program. It is not uncommon to see implementations of naïve algorithms achieve a higher FLOP rate than those of sophisticated ones. Often, the more advanced the algorithm, the more bookkeeping is required by the implementation and the smaller the fraction of the work that is spent doing actual floating point operations. Ideally, the execution time will still be lower since the total number of FLOPs will be significantly lower, too.⁴

Especially for a multi-threaded program, it is worthwhile to investigate how the FLOP rate evolves when assigning more cores to the program. While it is exceptionally unlikely to grow, it would ideally stay the same. If instead the FLOP rate drops rapidly as more cores are added, this indicates that the program is unable to distribute the work effectively across the cores.

Sometimes, a somewhat different measure is useful.

DEFINITION 6 (SPEEDUP) Let \mathcal{I} be the implementation of an algorithm where the execution time for an input of given size on a fixed number $c \in \mathbb{N}$ of cores is given by the cost function $t_c : \mathbb{N}_0 \rightarrow \mathbb{R}$. The quantity

$$S_c(n) = \frac{t_1(n)}{t_c(n)} \quad (2.6)$$

is called the **speedup** that is gained by giving more cores to \mathcal{I} .

Ideally, $S_c(n) = c$ for every $c \in \mathbb{N}$ and sufficiently large $n \in \mathbb{N}$. In practice, however, $S_c(n) \leq c$ due to additional communication overhead. In extreme cases,

⁴As a matter of fact, in this work we will later present an algorithm that has a higher FLOP count than another one but is still supposed to outperform it (achieve lower total execution time) due to the fact that under carefully contrived circumstances, it can make more efficient use of the hardware so to increase the FLOP rate to a level that compensates for the extra work.

it might even be observed that $S_c(n) < 1$ that is, the program actually runs *slower* if given more cores. This happens if the computation is basically sequential so all cores except one are waiting and the one that actually computes is further slowed down by additional coordination overhead.

2.3 Linear Systems

DEFINITION 7 (LINEAR SYSTEM) Let $n \in \mathbb{N}$ and $A \in \mathbb{R}^{n \times n}$. A **linear system** is an equation of the form

$$\mathbf{Ax} = \mathbf{b} \quad (2.7)$$

where A and $\mathbf{b} \in \mathbb{R}^n$ are given and $\mathbf{x} \in \mathbb{R}^n$ is to be determined.

The matrix A is called **coefficient matrix**.

If the coefficient matrix A in definition 7 is regular, then there always exists a $\mathbf{x} \in \mathbb{R}^n$ such that equation 2.7 becomes true. In the remainder of this work, we are only interested in linear systems where the coefficient matrix is regular.

Often times, many linear systems of the form $\mathbf{Ax}^{(k)} = \mathbf{b}^{(k)}$ should be solved for a single coefficient matrix $A \in \mathbb{R}^{n \times n}$ and multiple right-hand sides $\mathbf{b}^{(1)}, \dots, \mathbf{b}^{(k)} \in \mathbb{R}^n$ for $n, k \in \mathbb{N}$.

In the following sections, we will introduce some standard procedures for solving linear systems and discuss some of their properties.

2.3.1 LU Decomposition

LU decomposition is by far the most popular method for solving general linear systems. Given a regular coefficient matrix $A \in \mathbb{R}^{n \times n}$, a lower triangular matrix $L \in \mathbb{R}^{n \times n}$, an upper triangular matrix $U \in \mathbb{R}^{n \times n}$ and an orthogonal permutation matrix $P \in \{0, 1\}^{n \times n}$ are computed such that

$$LU = PA \quad (2.8)$$

Once this is done, the linear system

$$\mathbf{Ax} = \mathbf{b} \quad (2.9)$$

for $\mathbf{x} \in \mathbb{R}^n$ can be solved by multiplying equation 2.9 from the left with P and then substituting equation 2.8 for A to obtain

$$LU\mathbf{x} = P\mathbf{b} \quad (2.10)$$

Defining $\mathbf{c} = P\mathbf{b} \in \mathbb{R}^n$ and $U\mathbf{x} = \mathbf{y} \in \mathbb{R}^n$, we obtain the linear system

$$L\mathbf{y} = \mathbf{c} \quad (2.11)$$

Since L is a lower triangular matrix, \mathbf{c} is readily computed by means of *forward substitution* (§ 2.3.1.1). Substituting back into the definition of \mathbf{y} , we obtain yet another linear system

$$U\mathbf{x} = \mathbf{y} . \quad (2.12)$$

Since U is an upper triangular matrix, \mathbf{x} can now be computed by performing a *backward substitution* (§ 2.3.1.2) which gives us the solution for the original linear system in equation 2.9.

LU decomposition with pivoting (ie considering $P \neq \mathbf{1}$) for a $n \times n$ matrix requires about $n^3/3$ FLOPs [7]⁵ and is \mathcal{P} -complete [8]⁶.

The algebraic and numeric properties of the LU decomposition are extremely well understood and we will not further elaborate on them. Textbook knowledge may be found in Golub and Loan [7] or Press et al. [19] as well as innumerable other titles. We won't even provide the algorithm for obtaining the factors L and U or the permutation P since we don't use it in this work. The main reason we've introduced the LU decomposition at all is that it provides a convenient foundation for introducing the Cholesky decomposition (§ 2.3.2). However, we do provide algorithms for forward and backward substitution since these play an important role in our work.

2.3.1.1 Forward Substitution

Let $n, k \in \mathbb{N}$ and $L \in \mathbb{R}^{n \times n}$ be a regular lower triangular matrix. Given $\mathbf{c}^{(1)}, \dots, \mathbf{c}^{(k)} \in \mathbb{R}^n$, we want to find $\mathbf{y}^{(1)}, \dots, \mathbf{y}^{(k)} \in \mathbb{R}^n$ such that

$$L\mathbf{y}^{(l)} = \mathbf{c}^{(l)} \quad (2.13)$$

for all $l \in \{1, \dots, k\}$.

Solving

$$\mathbf{c}_i^{(l)} = \sum_{j=1}^i L_{ij}y_j^{(l)} = L_{ii}y_i^{(l)} + \sum_{j=1}^{i-1} L_{ij}y_j^{(l)} \quad (2.14)$$

for $y_i^{(l)}$, we get

$$y_i^{(l)} = \frac{1}{L_{ii}} \left(c_i^{(l)} - \sum_{j=1}^{i-1} L_{ij}y_j^{(l)} \right) \quad (2.15)$$

for all $l \in \{1, \dots, k\}$ and $i \in \{1, \dots, n\}$. Every result $y_i^{(l)}$ only depends on inputs and other $y_j^{(l)}$ with $j < i$ which means we can use equation 2.15 to compute $y_1^{(l)}$

⁵Note that Golub and Loan [7] define a FLOP as a single operation, not a pair of a multiplication and an addition, therefore, they obtain a value of $2n^3/3$ FLOPs.

⁶That is, there cannot be a parallelization scheme that solves the problem in sub-polynomial time while utilizing at most a polynomial (in n) number of computation units unless such a scheme exists for all problems in \mathcal{P} , the class of problems that can be solved sequentially in polynomial time.

and then in turn $y_2^{(l)}$ and so forth.⁷ It should be clear that the solutions for different l are all completely independent of each other. This leads to the straight-forward algorithm 2.2.

An analysis of algorithm 2.2 reveals that it performs about $k \frac{n^2}{2} \in \Theta(kn^2)$ FLOPs. What is more important for our work is that in the middle loop, every result depends on the previous one. Therefore, the loop cannot be fully parallelized which means that the algorithm has a critical path length on the order of $\Omega(n)$.

PROCEDURE ForwardSubstitution

INPUT

$L[1 \dots n][1 \dots n] : \text{REAL}$

$C[1 \dots n][1 \dots k] : \text{REAL}$

OUTPUT

$Y[1 \dots n][1 \dots k] : \text{REAL}$

VARIABLES

$i, j, l : \text{INTEGER}$

BEGIN

FOR $l \leftarrow 1$ TO k DO PARALLEL

FOR $i \leftarrow 1$ TO n DO

$Y_{il} \leftarrow C_{il}$

FOR $j \leftarrow 1$ TO $i - 1$ DO PARALLEL

$Y_{il} \leftarrow Y_{il} - L_{ij}Y_{jl}$

DONE

$Y_{il} \leftarrow Y_{il}/L_{ii}$

DONE

DONE

END

ALGORITHM 2.2: Algorithm for solving the multiple real linear systems $\mathbf{L}\mathbf{y}^{(l)} = \mathbf{c}^{(l)}$ with $\mathbf{c}^{(l)} \equiv C[:,l]$ and $\mathbf{y}^{(l)} \equiv Y[:,l]$ for $l \in \{1, \dots, k\}$ where \mathbf{L} is a regular lower triangular matrix.

2.3.1.2 Backward Substitution

Similar to the problem discussed in the previous section, the set of matrix equations

$$\mathbf{x}^{(l)}\mathbf{U} = \mathbf{b}^{(l)} \quad (2.16)$$

⁷Equation 2.15 is always well-formed because none of the L_{ii} for $i \in \{1, \dots, n\}$ can be zero or

$$\det(\mathbf{L}) = \prod_{i=1}^n L_{ii} = 0$$

in contradiction to our requirement that \mathbf{L} be regular.

– which have the same solutions as $U\mathbf{x}^{(l)} = \mathbf{y}^{(l)}$ ⁸ – with vectors $\mathbf{x}^{(l)}, \mathbf{b}^{(l)} \in \mathbb{R}^n$ and the regular upper triangular matrix $U \in \mathbb{R}^{n \times n}$ for $l \in \{1, \dots, k\}$ and $n, k \in \mathbb{N}$ can be solved for $\mathbf{x}^{(l)}$ by solving

$$\mathbf{b}_i^{(l)} = \sum_{j=i}^n x_j^{(l)} U_{ij} = x_i^{(l)} U_{ii} + \sum_{j=i+1}^n x_j^{(l)} U_{ij} \quad (2.17)$$

to obtain

$$x_i^{(l)} = \frac{1}{U_{ii}} \left(\mathbf{b}_i^{(l)} - \sum_{j=i+1}^n x_j^{(l)} U_{ij} \right). \quad (2.18)$$

This is implemented in algorithm 2.3 that also performs $k \frac{n^2}{2} \in \Theta(kn^2)$ FLOPs and has a critical path of length $\Omega(n)$ when executed in parallel.

PROCEDURE BackwardSubstitution

INPUT

$U[1 \dots n][1 \dots n] : \text{REAL}$

$B[1 \dots k][1 \dots n] : \text{REAL}$

OUTPUT

$X[1 \dots k][1 \dots n] : \text{REAL}$

VARIABLES

$i, j, l : \text{INTEGER}$

BEGIN

FOR $l \leftarrow 1$ TO k DO PARALLEL

FOR $i \leftarrow n$ DOWN TO 1 DO

$X_{li} \leftarrow B_{li}$

FOR $j \leftarrow i + 1$ TO n DO PARALLEL

$X_{li} \leftarrow X_{li} - U_{ij} X_{lj}$

DONE

$X_{li} \leftarrow X_{li} / U_{ii}$

DONE

DONE

END

ALGORITHM 2.3: Algorithm for solving the multiple real linear systems $\mathbf{x}^{(l)}U = \mathbf{b}^{(l)}$ with $\mathbf{b}^{(l)} \equiv \mathbf{B}[:,l]$ and $\mathbf{x}^{(l)} \equiv \mathbf{X}[:,l]$ for $l \in \{1, \dots, k\}$ where U is a regular upper triangular matrix.

2.3.2 Cholesky Decomposition

LU decomposition is a great black box algorithm because it works for arbitrary regular matrices. However, if we have additional information about the coefficient

⁸Remember (§ 2.1) that we don't distinguish between "row" and "column" vectors.

matrix, some work can be saved by exploiting these properties.

Let $n \in \mathbb{N}$ and $A \in \mathbb{R}^{n \times n}$ be a symmetric⁹ and positive definite¹⁰ matrix. (We will occasionally abbreviate the property “symmetric and positive definite” as “spd”.) Such a matrix is always regular.¹¹ Instead of decomposing A as in equation 2.8, we can compute a simpler decomposition of the form

$$A = LL^T \quad (2.19)$$

where $L \in \mathbb{R}^{n \times n}$ is a regular lower triangular matrix. This decomposition is known as *Cholesky*¹² or LL^T decomposition.

Once the factorization is computed, linear systems can be solved for any number of right hand sides via performing a forward (§ 2.3.1.1) followed by a backward substitution (§ 2.3.1.2) just as for the LU decomposition.

2.3.2.1 Inner Product Cholesky

A straight-forward formula for the Cholesky decomposition is readily derived by writing the factorization from equation 2.19 component wise as

$$\begin{pmatrix} L_{11} & 0 & \cdots & 0 \\ L_{21} & L_{22} & \ddots & \vdots \\ \vdots & \vdots & \ddots & 0 \\ L_{n1} & L_{n2} & \cdots & L_{nn} \end{pmatrix} \begin{pmatrix} L_{11} & L_{21} & \cdots & L_{n1} \\ 0 & L_{22} & \cdots & L_{n2} \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & L_{nn} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{21} & \cdots & A_{n1} \\ A_{21} & A_{22} & \cdots & A_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nn} \end{pmatrix}$$

and finding that

$$A_{ij} = \sum_{k=1}^j L_{ik}L_{kj} \quad (2.20)$$

for all $i \in \{1, \dots, n\}$, $j \in \{1, \dots, i\}$. (For $j > i$, simply use that A is symmetric, therefore $A_{ij} = A_{ji}$.) Equation 2.20 can be solved for L_{ij} to obtain

$$L_{ij} = \begin{cases} \sqrt{A_{ij} - \sum_{k=1}^{j-1} L_{ik}L_{jk}}, & j = i \\ \frac{1}{L_{jj}} (A_{ij} - \sum_{k=1}^{j-1} L_{ik}L_{jk}), & j < i \\ 0, & j > i \end{cases} \quad (2.21)$$

⁹Let $n \in \mathbb{N}$ and $A \in \mathbb{R}^{n \times n}$. The matrix A is *symmetric* if and only if $A_{ij} = A_{ji}$ for all $i, j \in \{1, \dots, n\}$.

¹⁰Let $n \in \mathbb{N}$ and $A \in \mathbb{R}^{n \times n}$. The matrix A is *positive definite* if $\lambda_i > 0$ for all of its eigenvalues $\lambda_1, \dots, \lambda_n$.

¹¹Let $n \in \mathbb{N}$ and $A \in \mathbb{R}^{n \times n}$ be spd. Then

$$\det(A) = \prod_{i=1}^n \lambda_i > 0$$

where $\lambda_1, \dots, \lambda_n$ are the positive (by definition) eigenvalues (repeated by their algebraic multiplicities) of A . Therefore, $\det(A) \neq 0$ so A is regular.

¹²In honor of the French mathematician André-Louis Cholesky (* 1875, Montguyon, France; † 1918, Bagnaux, France).

for all $i, j \in \{1, \dots, n\}$. By carefully looking at the dependencies in the above equation, one can see that if the entries in L are computed top-down and left-right, only values are used that have already been computed. This is implemented in algorithm 2.4; see Press et al. [19] for details.

Algorithm 2.4 performs $n^3/6$ FLOPs. This textbook version of the Cholesky decomposition only requires elementary linear algebra but, as written, has an outrageous quadratic critical path length. By rearranging the loops, we can derive an algorithm (§ 2.3.2.2) that only has a linear critical path length and a more desirable memory access pattern.

PROCEDURE CholeskyInnerProduct

INPUT

$A[1 \dots n][1 \dots n] : \text{REAL}$

OUTPUT

$L[1 \dots n][1 \dots n] : \text{REAL}$;; *may alias A for in-place update*

PRECONDITIONS

A is symmetric and positive definite

VARIABLES

$i, j, k : \text{INTEGER}$

$d : \text{REAL}$

BEGIN

FOR $i \leftarrow 1$ TO n DO

FOR $j \leftarrow 1$ TO i DO

$d \leftarrow A_{ij}$

FOR $k \leftarrow 1$ TO $j - 1$ DO PARALLEL

$d \leftarrow d - L_{ik}L_{jk}$

DONE

IF $i = j$ THEN

$L_{ij} \leftarrow \sqrt{d}$

ELSE

$L_{ij} \leftarrow d/L_{jj}$

FI

DONE

DONE

END

ALGORITHM 2.4: Algorithm for computing the Cholesky decomposition $LL^T = A$ of a symmetric and positive definite matrix $A \in \mathbb{N}^{n \times n}$. The essential step of this algorithm is the computation of the inner product in the innermost loop.

It would be admissible for A and L to refer to the same storage location since in no event, A_{ij} is loaded after L_{ij} has been stored. The structure of the matrix is implicit, that is, we don't ever look at the upper half of A nor do we ever write to the upper half of L (not even the zeros).

2.3.2.2 Gaxpy Cholesky

Golub and Loan [7] derive a version of the Cholesky decomposition that is based on so-called “gaxpy” operations.¹³

In equation 2.19, consider how the $j \in \{1, \dots, n\}$ -th column $A[:,j]$ of A is given by

$$A[:,j] = L[:,1:j](L^T[:,j]) = L[:,1:j]L[j,:]^T . \quad (2.23)$$

The matrix-vector product on the right side can be re-written as the vector sum

$$A[:,j] = \sum_{k=1}^j L[:,k]L[j,k]^T = \sum_{k=1}^j L_{jk}L[:,k] = L_{jj}L[:,j] + \sum_{k=1}^{j-1} L_{jk}L[:,k] \quad (2.24)$$

and this equation can be rearranged to

$$L_{jj}L[:,j] = \underbrace{A[:,j] - \sum_{k=1}^{j-1} L_{jk}L[:,k]}_{=: \mathbf{v}} . \quad (2.25)$$

The vector to the left is the j -th column of L scaled by the factor L_{jj} . If equation 2.25 should hold, v_j had better be L_{jj}^2 . Therefore,

$$L[:,j] = \frac{1}{\sqrt{v_j}} \mathbf{v} \quad (2.26)$$

gives us a formula for computing the the j -th column of L . Of course, $L[1:j-1][j]$ will always be zero. The square root is always real since L must be positive definite so $L_{jj} > 0$.

Equation 2.26 is implemented in algorithm 2.5. Like algorithm 2.4, it performs about $n^3/6$ FLOPS [7] but it only has a critical path of length $\Omega(n)$. This is a major improvement. In addition, the “gaxpy” operations can be implemented very efficiently on real hardware which makes this variant of the Cholesky factorization the algorithm of choice for parallel implementations [6].

¹³“Gaxpy” is a jargon term used by some people who have become (over-)used to the *Basic Linear Algebra Subprograms* (BLAS) to mean computing a vector as the sum of other vectors, each multiplied by a scalar factor. In BLAS, a vector \mathbf{y} is overwritten with $\alpha\mathbf{x} + \mathbf{y}$ where α is a scalar and \mathbf{x} is some other vector by calling one of the `_axpy` routines, to be read as “alpha times x plus y”. These overly terse function names were enforced by the draconian limits that early versions of the Fortran programming language set for the maximum allowed length of identifiers strings. The author admits that he is not entirely sure what the “g” in “gaxpy” is supposed to stand for.

As an important case for the following discussion, consider how the product of a matrix $A \in \mathbb{R}^{n \times m}$ and a vector $\mathbf{v} \in \mathbb{R}^m$ for $n, m \in \mathbb{N}$ can be computed via

$$A\mathbf{v} = \sum_{k=1}^m v_k A[:,k] . \quad (2.22)$$

```

PROCEDURE CholeskyGaxpy
INPUT
     $A[1 \dots n][1 \dots n] : \text{REAL}$ 
OUTPUT
     $L[1 \dots n][1 \dots n] : \text{REAL}$  ;; may alias A for in-place update
PRECONDITIONS
    A is symmetric and positive definite
VARIABLES
     $\mathbf{v}[1 \dots n] : \text{REAL}$ 
     $i, j, k : \text{INTEGER}$ 
BEGIN
    FOR  $j \leftarrow 1$  TO  $n$  DO
        FOR  $i \leftarrow j$  TO  $n$  DO PARALLEL
             $v_i \leftarrow A_{ij}$ 
        DONE
        FOR  $k \leftarrow 1$  TO  $j - 1$  DO PARALLEL
            FOR  $i \leftarrow j$  TO  $n$  DO PARALLEL
                 $v_i \leftarrow v_i - L_{jk}L_{ik}$ 
            DONE
        DONE
    DONE
END

```

ALGORITHM 2.5: Algorithm for computing the Cholesky decomposition $LL^T = A$ of a symmetric and positive definite matrix $A \in \mathbb{N}^{n \times n}$. The algorithm is dominated by the “gaxpy” operations that accumulate the vector \mathbf{v} .

It would be admissible for A and L to refer to the same storage location since in no event, A_{ij} is loaded after L_{ij} has been stored. The structure of the matrix is implicit, that is, we don’t ever look at the upper half of A nor do we ever write to the upper half of L (not even the zeros).

2.3.2.3 Outer Product Cholesky

There is yet another approach to the Cholesky decomposition that we'll only mention very briefly for the sake of completeness. Please refer to Golub and Loan [7] for details.

Let $\alpha = A_{11} > 0$, $\mathbf{a} = \mathbf{A}[2 : n][:]$ ¹⁴ $= \mathbf{A}[:, 2 : n]$ and $\tilde{\mathbf{A}} = \mathbf{A}[2 : n][2 : n]$, then

$$\mathbf{A} = \begin{pmatrix} \alpha & \langle \mathbf{a} | \\ | \mathbf{a} \rangle & \tilde{\mathbf{A}} \end{pmatrix} = \begin{pmatrix} \alpha^{1/2} & \langle \mathbf{0} | \\ \alpha^{-1/2} | \mathbf{a} \rangle & \mathbf{1} \end{pmatrix} \begin{pmatrix} \mathbf{1} & \langle \mathbf{0} | \\ | \mathbf{0} \rangle & \tilde{\mathbf{A}} - \alpha^{-1} | \mathbf{a} \rangle \langle \mathbf{a} | \end{pmatrix} \begin{pmatrix} \alpha^{1/2} & \alpha^{-1/2} \langle \mathbf{a} | \\ | \mathbf{0} \rangle & \mathbf{1} \end{pmatrix} \quad (2.27)$$

where the first equality simply states the definition and the second can easily be proved by multiplying out. The matrix $\mathbf{A}^{(1)} = \tilde{\mathbf{A}} - \alpha^{-1} | \mathbf{a} \rangle \langle \mathbf{a} | \in \mathbb{R}^{(n-1) \times (n-1)}$ is again symmetric and positive definite so if $\mathbf{L}^{(1)} (\mathbf{L}^{(1)})^T = \mathbf{A}^{(1)}$ is a Cholesky factorization of $\mathbf{A}^{(1)}$, then

$$\mathbf{L} = \begin{pmatrix} \alpha^{1/2} & \langle \mathbf{0} | \\ \alpha^{-1/2} | \mathbf{a} \rangle & \mathbf{L}^{(1)} \end{pmatrix}. \quad (2.28)$$

Therefore, after $n - 1$ steps of recursive application, we have the Cholesky decomposition $\mathbf{L}\mathbf{L}^T = \mathbf{A}$ of \mathbf{A} .

This approach also performs $n^3/6$ FLOPs [7] and has a critical path of $\Omega(n)$. However, the outer product Cholesky decomposition has a less favorable memory access pattern compared with the ‘‘gaxpy’’ version which makes it a little less appealing in practice [7, 6].

2.3.3 Block LU Decomposition

In section 2.3.1 we have introduced the general LU decomposition and then elaborated on the important case that the coefficient matrix is symmetric and positive definite in section 2.3.2. In this section, we'll present a variant of the LU factorization that constitutes the fundamental building block in our contribution. We discuss it for the general LU decomposition here but it can likewise be applied to the Cholesky decomposition with the additional simplification that a few computations become obsolete thanks to symmetry.

Let $n \in \mathbb{N}$ and $\mathbf{M} \in \mathbb{R}^{n \times n}$ be regular. Now pick a $k \in \{1, \dots, n\}$ and block \mathbf{M} into $\mathbf{M}_{11} = \mathbf{M}[1 : k][1 : k]$, $\mathbf{M}_{12} = \mathbf{M}[1 : k][k : n]$, $\mathbf{M}_{21} = \mathbf{M}[k : n][1 : k]$ and $\mathbf{M}_{22} = \mathbf{M}[k : n][k : n]$ where we shall assume that \mathbf{M}_{11} is regular¹⁵. If $\mathbf{L}_{11}\mathbf{U}_{11} = \mathbf{M}_{11}$ with a regular lower triangular matrix $\mathbf{L}_{11} \in \mathbb{R}^{k \times k}$ and a regular upper triangular matrix $\mathbf{U}_{11} \in \mathbb{R}^{k \times k}$ is a LU decomposition of \mathbf{M}_{11} , then we can perform $n - k$ forward substitutions to solve $\mathbf{L}_{11}\mathbf{U}_{12} = \mathbf{M}_{12}$ for $\mathbf{U}_{12} \in \mathbb{R}^{k \times (n-k)}$

¹⁴Remember (§ 2.1) that we do not distinguish between row and column vectors.

¹⁵A sufficient condition for \mathbf{M}_{11} to be regular is that \mathbf{M} has an LU factorization without pivoting (pivoting). This should be clear from the fact that the LU decomposition (without permutations) of a regular matrix, if one exists, is unique.

and $n - k$ backward substitutions to solve $L_{21}U_{11} = M_{21}$ for $L_{21} \in \mathbb{R}^{(n-k) \times k}$. Introducing $\tilde{M} = M_{22} - L_{21}U_{12}$, we get

$$M = \begin{pmatrix} M_{11} & M_{12} \\ M_{21} & M_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & \mathbf{0} \\ L_{21} & \mathbf{1} \end{pmatrix} \begin{pmatrix} \mathbf{1} & \mathbf{0} \\ \mathbf{0} & \tilde{M} \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} \\ \mathbf{0} & \mathbf{1} \end{pmatrix} \quad (2.29)$$

which can be easily proven by multiplying out. The matrix $\tilde{M} \in \mathbb{R}^{(n-k) \times (n-k)}$ is sometimes called the *Schur complement*¹⁶ of M . If $L_{22}U_{22} = \tilde{M}$ with a regular lower triangular matrix $L_{22} \in \mathbb{R}^{(n-k) \times (n-k)}$ and regular upper triangular matrix $U_{22} \in \mathbb{R}^{(n-k) \times (n-k)}$ is the LU decomposition of \tilde{M} , then

$$\begin{pmatrix} L_{11} & \mathbf{0} \\ L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} \\ \mathbf{0} & U_{22} \end{pmatrix} = \begin{pmatrix} M_{11} & M_{12} \\ M_{21} & M_{22} \end{pmatrix} \quad (2.30)$$

is the LU decomposition of M . This is again proved by simply multiplying out.

Therefore, the factorization of M can be computed recursively. This approach to the LU decomposition does not change (for dense matrices) the amount of FLOPs required but makes matrix products become the dominant operation [7] which is good because it can be fully parallelized and there are highly optimized implementations available for it. The overall critical path length of the algorithm remains on the order of $\Omega(n)$, however. Once again, please refer to Golub and Loan [7] for details.¹⁷

2.3.4 Block LDU Decomposition

The block LU decomposition can be modified to yield a decomposition into three factors.

Let $n \in \mathbb{N}$ and $M \in \mathbb{R}^{n \times n}$ be regular. Let further $k \in \{1, \dots, n\}$ and $M_{11} = M[1 : k][1 : k]$ be regular as well¹⁸. Equation 2.29 can be modified to

$$M = \begin{pmatrix} M_{11} & M_{12} \\ M_{21} & M_{22} \end{pmatrix} = \begin{pmatrix} \mathbf{1} & \mathbf{0} \\ L_{21} & \mathbf{1} \end{pmatrix} \begin{pmatrix} M_{11} & \mathbf{0} \\ \mathbf{0} & \tilde{M} \end{pmatrix} \begin{pmatrix} \mathbf{1} & U_{12} \\ \mathbf{0} & \mathbf{1} \end{pmatrix} \quad (2.31)$$

with $L_{21} = M_{21}M_{11}^{-1}$, $U_{12} = M_{11}^{-1}M_{12}$ and $\tilde{M} = M_{22} - M_{21}M_{11}^{-1}M_{12}$. Note that the definitions of L_{21} , U_{12} and \tilde{M} are exactly the same as for equation 2.29. The only difference is really that M_{11} is not factored.

2.3.5 Matrix Inversion

DEFINITION 8 (INVERSE MATRIX) Let $n \in \mathbb{N}$ and $M \in \mathbb{R}^{n \times n}$ be regular. The matrix $M^{-1} \in \mathbb{R}^{n \times n}$ such that

$$MM^{-1} = \mathbf{1} . \quad (2.32)$$

¹⁶In honor of the mathematician Issai Schur (* 1875, Mogilev, Russian Empire; † 1941, Tel Aviv, then Mandatory Palestine) who spent most of his working life in Germany.

¹⁷Unfortunately, there is a typo in § 3.2.10 of our copy of Golub and Loan [7] where they erroneously write \tilde{A} instead of 1_{n-r} (in their typographical conventions).

¹⁸See footnote 15 on page 19.

is called the **inverse** of M .

If $n, k \in \mathbb{N}$, $M \in \mathbb{R}^{n \times n}$ and $\mathbf{b}_1, \dots, \mathbf{b}_k \in \mathbb{R}^n$, then we can solve the linear systems

$$M\mathbf{x}_l = \mathbf{b}_l$$

for $\mathbf{x}_1, \dots, \mathbf{x}_k \in \mathbb{R}^n$ by computing the inverse matrix $M^{-1} \in \mathbb{R}^{n \times n}$ and then using

$$\mathbf{x}_i = M^{-1}\mathbf{b} \quad (2.33)$$

for $l \in \{1, \dots, k\}$.

This matrix-vector product requires n^2 FLOPs and has a critical path of length $\Omega(\log(n))$.

2.3.5.1 Gauß-Jordan Elimination

The textbook algorithm for computing the inverse of a regular matrix – known as *Gauß-Jordan Elimination*¹⁹ with partial pivoting – constructs the inverse matrix by applying a sequence of row transformations to the matrix and the identity matrix. After the original matrix has been transformed into the identity matrix, the identity matrix has been transformed into the inverse matrix. Since neither identity matrix needs not be stored explicitly, the process can be done in-place, successively overwriting the original matrix with its inverse as it goes. Please refer to Press et al. [19] for details.

If the right hand sides are known in advance, the Gauß-Jordan algorithm can be adapted to solve the linear systems as it inverts the matrix. This is preferred over inverting the matrix first and then multiplying with the right sides since it has better numerical stability. [19] We are not using this in our work so we've merely mentioned it.

The algorithm performs n^3 FLOPs (three times as much as a straight-forward *LU* decomposition) and has a critical path on the order of $\Omega(n)$. If the matrix is known to be symmetric and positive definite, some work can be saved since no pivoting will be needed and only half of the matrix needs to be computed.²⁰

2.3.5.2 Strassen Inversion

In 1969, Strassen²¹ presented a way to compute the product of two $n \times n$ matrices with $n = 2^k$ for $k \in \mathbb{N}$ using only $O(n^\omega)$ FLOPs where $\omega \leq \log_2(7) \approx 2.8$ [21].

¹⁹In honor of the German mathematician Johann Carl Friedrich Gauss (* 1777, Brunswick, then part of the Holy Roman Empire; † 1855, Göttingen, then Kingdom of Hanover) and the German geodesist Wilhelm Jordan (* 1842, Ellwangen, then Kingdom of Württemberg; † 1899, Hanover, then a province of Prussia).

²⁰The inverse of a symmetric matrix is symmetric. Proof: Let $n \in \mathbb{N}$ and $M \in \mathbb{R}^{n \times n}$ be regular and symmetric, then $MM^{-1} = \mathbf{1}$. Transpose on both sides to get $(MM^{-1})^T = \mathbf{1}^T$ and further $M^{-T}M^T = \mathbf{1}^T$ which – since both, M and $\mathbf{1}$ are symmetric – is equivalent to $M^{-T}M = \mathbf{1}$. Now multiply from the right with M^{-1} and obtain $M^{-T}MM^{-1} = \mathbf{1}M^{-1}$ equivalent to $M^{-T} = M^{-1}$ which was to be shown.

²¹Volker Strassen (* 1936, Düsseldorf-Gerresheim, Germany) is a German mathematician.

The naïve approach takes $\Theta(n^3)$ FLOPs. Since then, much research effort was invested in lowering the upper bound for ω . The current-best (as of 2014) result is $\omega \leq 2.3728639$ [12] but it is widely suspected that for every $\epsilon > 0$ an algorithm with $\omega < 2 + \epsilon$ exists [1]. As already mentioned by Strassen, whatever the bound on ω for matrix-matrix multiplication is, matrix inversion can be done with the same asymptotic complexity.

Let $k \in \mathbb{N}$ and $A \in \mathbb{R}^{n \times n}$ with $n = 2^k$ be regular and have an LU factorization without permuting. Then block A and A^{-1} into $2^{k-1} \times 2^{k-1}$ sub-matrices

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \quad \text{and} \quad A^{-1} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} \quad (2.34)$$

and compute C_{11} , C_{12} , C_{21} and C_{22} via

$$S_I = A_{11}^{-1} \quad (2.35)$$

$$S_{II} = A_{21} S_I \quad (2.36)$$

$$S_{III} = S_I A_{12} \quad (2.37)$$

$$S_{IV} = A_{21} S_{III} \quad (2.38)$$

$$S_V = S_{IV} - A_{22} \quad (2.39)$$

$$S_{VI} = S_V^{-1} \quad (2.40)$$

$$C_{12} = S_{III} S_{VI} \quad (2.41)$$

$$C_{21} = S_{VI} S_{II} \quad (2.42)$$

$$S_{VII} = S_{III} C_{21} \quad (2.43)$$

$$C_{11} = S_I - S_{VII} \quad (2.44)$$

$$C_{22} = -S_{VI} \quad (2.45)$$

where the inversions in step 2.35 and 2.40 enter the algorithm recursively until $n = 1$ (ie $k = 0$) when matrix inversion degenerates to scalar inversion²². [21]

Equations 2.35 to 2.45 can be combined into the closed expression

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}^{-1} = \begin{pmatrix} A_{11}^{-1} + A_{11}^{-1} A_{12} \tilde{S}^{-1} A_{21} A_{11}^{-1} & -A_{11}^{-1} A_{12} \tilde{S}^{-1} \\ -\tilde{S}^{-1} A_{21} A_{11}^{-1} & \tilde{S}^{-1} \end{pmatrix} \quad (2.46)$$

with $\tilde{S} = A_{22} - A_{21} A_{11}^{-1} A_{12}$. This is quite similar to the blocking schemes we have already seen (§§ 2.3.4, 2.3.2.3) and can also be proved simply by multiplying out, except that it takes more ink and paper this time. We mention this to signify that the approach works for any matrix size n , not only powers of two.

²²Alternatively, the recursion is aborted at a larger fixed $n \in \mathbb{N}$ at which point a traditional inversion algorithm (eg Gauß-Jordan, § 2.3.5.1) is used to invert the sub-matrices. This doesn't affect the asymptotic complexity but can greatly reduce the computational overhead in practice. See § 2.3.5.4 for an alternative to a fixed-size limit.

If \mathbf{A} is symmetric, equation 2.46 can be somewhat simplified by using that $\mathbf{A}_{12} = \mathbf{A}_{21}^T$ as well as $\mathbf{A}_{11}^{-T} = \mathbf{A}_{11}^{-1}$ and $\tilde{\mathbf{S}}^{-T} = \tilde{\mathbf{S}}^{-1}$.

Strassen inversion of an $n \times n$ matrix for $n \in \mathbb{N}$ performs $\Theta(n^\omega)$ FLOPs where ω is the exponent for matrix-matrix multiplication and has a critical path length bounded by $\Omega(n)$. [20]

2.3.5.3 Newton Inversion

Unlike the previously discussed direct algorithms, the algorithm discussed in this section is iterative. That is, it starts with an initial guess for the inverse that is refined iteratively in each step. Once the current result is “good enough”, the iteration is aborted. This approach will always yield an inexact result, not only because of numeric round-off errors and instabilities but due to the very nature of the procedure.

Newton’s²³ general root finding algorithm can also be applied to matrix inversion. Let $\mathbf{A} \in \mathbb{R}^{n \times n}$ for $n \in \mathbb{N}$ be regular. Given an initial guess $\mathbf{X}^{(0)}$ on the inverse \mathbf{A}^{-1} , the sequence

$$\mathbf{X}^{(k+1)} = 2\mathbf{X}^{(k)} - \mathbf{X}^{(k)}\mathbf{A}\mathbf{X}^{(k)} \quad (2.47)$$

converges quadratically to $\mathbf{X}^{(k)} \xrightarrow{k \rightarrow \infty} \mathbf{A}^{-1}$ if the norm of the residual

$$\|\mathbf{R}^{(0)}\| = \|\mathbf{1} - \mathbf{X}^{(0)}\mathbf{A}\| < 1 \quad (2.48)$$

Pan and Reif [16] have shown that this is guaranteed if $\mathbf{X}^{(0)}$ is chosen as

$$\mathbf{X}^{(0)} = \lambda^{-1}\mathbf{A}^T \quad \text{with} \quad \lambda = \|\mathbf{A}\|\|\mathbf{A}^T\| \quad (2.49)$$

for an arbitrary matrix norm $\|\cdot\|$. For example,

$$\lambda = \sum_{i=1}^n \sum_{j=1}^n A_{ij}^2 \quad (2.50)$$

If \mathbf{A} is symmetric and positive definite,

$$\mathbf{X}^{(0)} = \lambda^{-1}\mathbf{1} \quad \text{with} \quad \lambda = \|\mathbf{A}\| \quad (2.51)$$

is an appropriate choice, too [20].

For more details, please refer to Press et al. [19] for general information or Pan and Reif [16, 17] for an in-depth discussion.

Algorithm 2.6 shows a high-level implementation of the process. Note that all steps can be fully parallelized. Pan and Reif show that if $\mathbf{X}^{(0)}$ is chosen like they say, the iteration will converge after $O(\log(n))$ steps for a fixed tolerated residual $\epsilon > 0$. Therefore, the algorithm achieves a critical path length of $\Omega(\log(n)^2)$. [16]

²³In honor of the British scientist Sir Isaac Newton (* 1643, Woolsthorpe, then England; † 1727, Kensington, Great Britain).

The amount of work to be done in each iteration is dominated by computing matrix-matrix products and therefore on the order of $\Theta(n^\omega)$. Combined with the bound for the number of iterations, we get an estimated number of FLOPs on the order of $O(n^\omega \log(n))$ [16]. Note that if the algorithm is chosen for its critical path, plugging a matrix multiplication algorithm that destroys this path length is probably not desirable so a more conservative bound for the number of FLOPs might be appropriate [20].

```

PROCEDURE InverseNewton
INPUT
   $A[1 \dots n][1 \dots n] : \text{REAL}$ 
OUTPUT
   $A^{-1}[1 \dots n][1 \dots n] : \text{REAL}$ 
PRECONDITIONS
   $A$  is regular
POSTCONDITIONS
   $A^{-1}$  is an approximate inverse of  $A$ 
CONSTANTS
   $\delta_{\text{tol}} : \text{REAL} \leftarrow$  tolerance for refinement
   $\epsilon_{\text{tol}} : \text{REAL} \leftarrow$  tolerance for residual
VARIABLES
   $D[1 \dots n][1 \dots n] : \text{REAL}$ 
   $\lambda : \text{REAL}$ 
BEGIN
   $\lambda \leftarrow \|A\| \|A^T\|$ 
   $A^{-1} \leftarrow A^T / \lambda$ 
  REPEAT DO
     $D \leftarrow A^{-1} A$ 
     $A^{-1} \leftarrow (2 \mathbf{1} - D) A^{-1}$ 
  WHILE  $\|D\| \geq \delta_{\text{tol}}$  DONE
  IF  $\|\mathbf{1} - A^{-1} A\| \geq \epsilon_{\text{tol}}$  THEN
    ERROR desired accuracy not reached
  FI
END

```

ALGORITHM 2.6: High-level algorithm for inverting a matrix A iteratively using Newton's method and the starting matrix suggested by Pan and Reif [16]. Those who feel lucky enough might hard-code a fixed number of iterations instead of using an adaptive stopping criterion. The norm $\|\cdot\|$ could be any convenient matrix norm.

2.3.5.4 The NeSt Algorithm

Sanders, Speck, and Steffen have combined Strassen Inversion (§ 2.3.5.2) and Newton’s algorithm (§ 2.3.5.3) to obtain a work-efficient matrix inversion algorithm – called “NeSt” for “Newton & Strassen” – for symmetric and positive definite matrices²⁴ with poly-logarithmic time complexity [20]. This algorithm forms the foundation of our contribution.

Given a fixed number $p \in \mathbb{N}$ of computing cores and a sufficiently large $n \in \mathbb{N}$, a symmetric and positive definite input matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ is split into smaller matrices by applying Strassen’s algorithm $h \in \{0, \dots, \lfloor \log_2(n) \rfloor\}$ times. Each of these sub-matrices is then inverted using Newton’s algorithm.

To increase the numerical stability of the algorithm, the result of Strassen’s algorithm can further be processed by a single Newton iteration at each recursive step. This only increases the algorithm’s complexity by a constant factor (since Strassen’s algorithm requires matrix multiplications anyway). [20]

The choice of the recursion parameter h allows tuning the algorithm between the two extremes of

- pure Newton inversion ($h = 0$) providing maximum parallelization at the cost of a logarithmic factor of additional work and
- pure Strassen inversion ($h = \lfloor \log_2(n) \rfloor$) providing maximum efficiency (depending only on the fast matrix multiplication exponent ω) at the price of a critical path length on the order of $\Omega(n)$.

Sanders, Speck, and Steffen show that for any $\epsilon > 0$, if the recursion depth is chosen as

$$h = (1 + \epsilon) \frac{\log(\log(n))}{\omega - 1} \quad (2.53)$$

then NeSt performs $\frac{4}{2^{\omega-2}} + C$ (where C is a small constant) times the work and has a critical path length bounded by $O\left(\log(n)^{\frac{\omega+\epsilon}{\omega-1}}\right)$ times the critical path length of the plugged matrix-matrix multiplication algorithm. For the naïve algorithm with $\omega = 3$ and a critical path length of $\Theta(\log(n))$ this gives $(1 + C)n^3$ FLOPs and critical path of $\Theta\left(\log(n)^{5/2+\epsilon}\right)$. [20] More informally, h should be chosen small enough to keep all p computing cores busy but large enough to keep the total amount of work small.

Equipped With these algorithmic building blocks, we are now turning to the problem we have studied and present our own contribution.

²⁴It should become clear from the following discussion that the concept is easily extended to matrices that are not symmetric and positive definite but do have an LU factorization without permuting. On the other hand, there is a more general solution, realizing that for every regular matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ and $n \in \mathbb{N}$, the matrix $\mathbf{B} := \mathbf{A}^T \mathbf{A}$ is symmetric and positive definite and

$$\mathbf{A}^{-1} = \mathbf{A}^{-1} \mathbf{1} = \mathbf{A}^{-1} (\mathbf{A}^{-T} \mathbf{A}^T) = (\mathbf{A}^{-1} \mathbf{A}^{-T}) \mathbf{A}^T = (\mathbf{A}^T \mathbf{A})^{-1} \mathbf{A}^T = \mathbf{B}^{-1} \mathbf{A}^T. \quad (2.52)$$

See Sanders, Speck, and Steffen [20] for more information.

```

PROCEDURE InverseNeSt
INPUT
     $h$  : INTEGER
     $A[1 \dots n][1 \dots n]$  : REAL
OUTPUT
     $A^{-1}[1 \dots n][1 \dots n]$  : REAL
PRECONDITIONS
     $A$  is symmetric and positive definite
POSTCONDITIONS
     $A^{-1}$  is an approximate inverse of  $A$ 
CONSTANTS
     $\omega$  : REAL  $\leftarrow$  fast matrix–matrix multiplication exponent
     $\epsilon$  : REAL  $\leftarrow$  a small positive constant
VARIABLES
     $A_{11}^{-1}[1 \dots n/2][1 \dots n/2]$  : REAL
     $\tilde{S}^{-1}[1 \dots n - n/2][1 \dots n - n/2]$  : REAL
BEGIN
    IF  $h < 0$  THEN  $h \leftarrow \lfloor (1 + \epsilon) \frac{\log(\log(n))}{\omega - 1} \rfloor$  FI ;; provide default
    IF  $h = 0$  THEN
        CALL InverseNewton( $A, A^{-1}$ )
    ELSE
        ;; Invert the sub–matrices.
        CALL InverseNeSt( $h - 1, A_{11}, A_{11}^{-1}$ )
        CALL InverseNeSt( $h - 1, A_{22} - A_{21}A_{11}^{-1}A_{12}, \tilde{S}^{-1}$ )
        ;; Combine the results.
         $A_{11}^{-1} \leftarrow A_{11}^{-1} + A_{11}^{-1}A_{12}\tilde{S}^{-1}A_{21}A_{11}^{-1}$ 
         $A_{12}^{-1} \leftarrow -A_{11}^{-1}A_{12}\tilde{S}^{-1}$ 
         $A_{21}^{-1} \leftarrow -\tilde{S}^{-1}A_{21}A_{11}^{-1}$ 
         $A_{22}^{-1} \leftarrow \tilde{S}^{-1}$ 
        ;; Perform a single Newton iteration as stabilization (optional).
         $A^{-1} \leftarrow (2\mathbf{1} - A^{-1}A)A^{-1}$ 
    FI
END

```

ALGORITHM 2.7: The NeSt algorithm for matrix inversion. If called with $h < 0$, the theoretical optimum is substituted as default. The code is of course to be taken with a grain of salt. It is intended to give the basic idea of the algorithm but even ignores obvious optimization like common sub-expression elimination. It also ignores the fact that A and A^{-1} are symmetric. For better readability and to make it fit inside the page's margins, the above listing uses the shorthand aliases $A_{11} := A[1 : \frac{n}{2}][1 : \frac{n}{2}]$, $A_{12} := A[1 : \frac{n}{2}][\frac{n}{2} + 1 : n]$, $A_{21} := A[\frac{n}{2} + 1 : n][1 : \frac{n}{2}]$, $A_{22} := A[\frac{n}{2} + 1 : n][\frac{n}{2} + 1 : n]$ and likewise $A_{11}^{-1} := A^{-1}[1 : \frac{n}{2}][1 : \frac{n}{2}]$, $A_{12}^{-1} := A^{-1}[1 : \frac{n}{2}][\frac{n}{2} + 1 : n]$, $A_{21}^{-1} := A^{-1}[\frac{n}{2} + 1 : n][1 : \frac{n}{2}]$, $A_{22}^{-1} := A^{-1}[\frac{n}{2} + 1 : n][\frac{n}{2} + 1 : n]$.

2.4 Hierarchical Matrices

DEFINITION 9 (HIERARCHICAL MATRIX) A **hierarchical matrix** (H-matrix) is a matrix that is represented as a cluster tree of sub-matrices.

At the lowest level, the sub-matrices of an H-matrix may be represented by any suitable matrix format, such as a full (dense) matrix, a compressed sparse matrix format or a special low-rank representation [4].

The H-matrix representation is worthwhile especially if the structure of a matrix is known a priori to consist of blocks with non-zero elements and large zero blocks in between.

As an example, consider the following matrix

$$M = \begin{array}{|c|cc|} \hline \begin{array}{|c|} \hline A \\ \hline \end{array} & \begin{array}{|c|} \hline 0 \\ \hline \end{array} & \begin{array}{|c|c|} \hline 0 & 0 \\ \hline B & 0 \\ \hline \end{array} \\ \hline & \begin{array}{|c|} \hline 0 \\ \hline \end{array} & \begin{array}{|c|} \hline 0 \\ \hline \end{array} \\ \hline \begin{array}{|c|} \hline 0 \\ \hline \end{array} & \begin{array}{|c|} \hline C \\ \hline \end{array} & \begin{array}{|c|} \hline 0 \\ \hline \end{array} \\ \hline & \begin{array}{|c|} \hline 0 \\ \hline \end{array} & \begin{array}{|c|} \hline D \\ \hline \end{array} \\ \hline \end{array} \quad (2.54)$$

where we know that the blank blocks are all-zero but don't know much about the inner structure of the shaded blocks.

Storing M as a plain full matrix is not optimal. Not only would we waste a lot of memory for storing all the redundant zeros, we would also throw away the structural information about M . Once this information is lost by choosing an inappropriate data structure, it can no longer be exploited in subsequent algorithms that operate on the matrix without rediscovering it first (potentially at a high price).

On the other hand, storing M in a general sparse matrix format wouldn't be optimal either. Compressed matrix formats avoid saving the zeros but this comes at the price of (usually) tripled storage requirements for the non-zero elements.²⁵ Also, iterating over the non-zero entries will incur constant overhead. If the matrix is very sparse, this will be worthwhile. However, while M has large zero blocks, it is not particularly sparse. Also, as for the full matrix representation, the sparse format loses the structural information.

As the graphic already suggests, an attractive data structure for M might be based on quad-trees. If a sub-tree consists only of zeros, we need not store it at all but instead represent it by a special NIL value. If a sub-tree consists only of non-zero blocks, we store it using a classical full or sparse matrix format.²⁶ If a

²⁵This is plausible since effectively, most of these formats somehow (usually implicitly) have to store the row and column index along with each entry.

²⁶Many articles on H-matrices also discuss storing blocks (especially those with a low rank) in a decomposed form. Since we don't need this for our application, we will not further discuss that possibility.

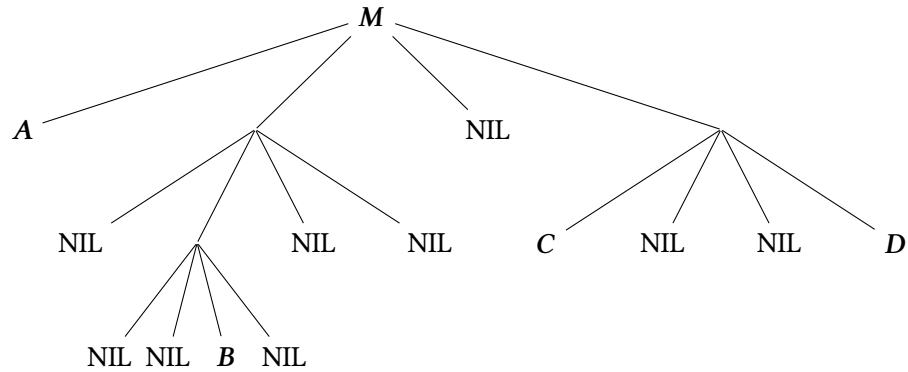


FIGURE 2.1: The hierarchical matrix M from equation 2.54 represented as a quad-tree.

sub-tree contains both, zero and non-zero blocks, we divide it further until we reach one of the above two cases. Obviously, since the number of elements in a matrix is finite and a single element is always either zero or non-zero in a trivial sense, this procedure always terminates. The representation of M as a quad-tree is sketched in figure 2.1.

H-matrices need not be represented as quad-trees, if there is another tree format that is more appropriate to model the inner structure of the matrix. In the next section, we will investigate a class of matrices that are elegantly represented as sept-trees.

2.4.1 Nested Dissection

Discovering the structure of an H-matrix can be a challenging task; especially if permuting rows and columns should be considered in order to obtain blocks as big as possible. Fortunately, for some applications, the structure appears naturally as part of the problem.

Every square matrix $M \in \mathbb{R}^{n \times n}$ with $n \in \mathbb{N}$ can be interpreted as the adjacency matrix of a conductivity graph $G = (V, E, \omega)$ with vertices $V = \{1, \dots, n\}$, edges $E = \{(i, j) \in (V \times V) : M_{ij} \neq 0\}$ and edge weight function

$$\begin{aligned} \omega : (V \times V) &\rightarrow \mathbb{R} \\ (i, j) &\mapsto M_{ij} . \end{aligned}$$

DEFINITION 10 (ROW & COLUMN INDEX) Let $n \in \mathbb{N}$ and $G = (V, E, \omega)$ be an edge-weighted graph with n vertices V , edges $E \subseteq (V \times V)$ and edge weight function $\omega : (V \times V) \rightarrow \mathbb{R}$. Let further $A \in \mathbb{R}^{n \times n}$ be an adjacency matrix of G .

The functions $\text{row}_A : V \rightarrow \{1, \dots, n\}$ and $\text{col}_A : V \rightarrow \{1, \dots, n\}$ shall map each vertex to its respective row and column index in A . That is

$$\forall u, v \in V : A_{\text{row}_A(u), \text{col}_A(v)} = \omega((u, v)) . \quad (2.55)$$

Consider now the case where G is a regular conductivity grid. In such a grid, every vertex has a constant number of neighbors. Importantly, the average degree (number of neighbors) of a vertex does not scale with the size of the grid. It is also the case that vertices are only adjacent to vertices “close” to them.²⁷ It is therefore trivial to obtain a good partitioning of the graph, simply by cutting the grid into two equally large halves $\Pi_A \subset V$ and $\Pi_B \subset V$ with $\Pi_A \cup \Pi_B = V$ and $\Pi_A \cap \Pi_B = \emptyset$. The vertices can then be classified into three categories.

- The vertices

$$V_A = \{v \in \Pi_A : \forall (u, w) \in E : v \notin \{u, w\} \vee u \in \Pi_A \wedge w \in \Pi_A\} \quad (2.56)$$

in Π_A that have only neighbors in Π_A ,

- the vertices

$$V_B = \{v \in \Pi_B : \forall (u, w) \in E : v \notin \{u, w\} \vee u \in \Pi_B \wedge w \in \Pi_B\} \quad (2.57)$$

in Π_B that have only neighbors in Π_B and

- the remaining vertices

$$V_C = \{v \in V : \exists (u, w) \in E : v \in \{u, w\} \wedge (u \in \Pi_A \Leftrightarrow w \in \Pi_B)\} \quad (2.58)$$

that have incident intra-cluster edges.

We define $n_A = |V_A|$, $n_B = |V_B|$ and $n_C = |V_C|$. The rows and columns of M are now permuted to yield \tilde{M} such that for each vertex $v \in V$

$$\text{row}_{\tilde{M}}(v) = \text{col}_{\tilde{M}}(v) \quad (2.59)$$

and

$$v \in V_A \Rightarrow 1 \leq \text{row}_{\tilde{M}}(v) \leq n_A \quad (2.60)$$

$$v \in V_B \Rightarrow n_A < \text{row}_{\tilde{M}}(v) \leq n_A + n_B \quad (2.61)$$

$$v \in V_C \Rightarrow n_A + n_B < \text{row}_{\tilde{M}}(v) \leq n . \quad (2.62)$$

We obtain a matrix with seven non-zero blocks.

$$\tilde{M} = \begin{pmatrix} A & \mathbf{0} & X \\ \mathbf{0} & B & Y \\ U & V & C \end{pmatrix} \quad (2.63)$$

block	origin	destination
A	V_A	V_A
B	V_B	V_B
C	V_C	V_C
U	V_C	V_A
V	V_C	V_B
X	V_A	V_C
Y	V_B	V_A

TABLE 2.1: After one step of nested dissection, the permuted matrix \tilde{M} contains seven non-zero blocks (see eq 2.63) that hold the weights for specific edges (u, v) in the graph. This table shows for each of the blocks into what partition the vertices u (origin) and v (destination) belong. The sets V_A , V_B and V_C are defined in equations 2.56 to 2.58.

Table 2.1 details how these blocks are to be interpreted.

The procedure is now repeated recursively on the sub-graphs induced by V_A and V_B . The required permutations never destroy the structure on the higher levels. This procedure, known as *nested dissection*, is shown graphically in figure 2.2.

The appealing data structure to represent such a matrix is a tree where each node has five children for the respective blocks and it will be implicit what sub-matrix is represented by each child.

2.4.1.1 Symmetry

If the matrix M is *symmetric*, the graph G can be replaced by an *undirected* edge-weighted graph G' and the procedure carried out as above. Note that in this case, the blocks **A**, **B** and **C** will again be symmetric and $X = U^T$ as well as $Y = V^T$. A smart data structure would then represent the H-matrix as a quint-tree with a special representation for the diagonal blocks that only stores the lower half of the blocks **A**, **B** and **C**. Access to the upper half of the matrix would then either be prohibited or mapped to the equivalent lower part.

If the matrix comes from a finite-element problem where “conductivity” means some real physical quantity, it is not unlikely for the matrix to be symmetric since many physical quantities are undirected (such as thermal conductivity).

²⁷This is the opposite of a “small world graph” (often found in social networks) where the diameter of the graph only grows logarithmically in the number of vertices even though the average node degree is also constant. This is due to few but important distant links that connect remote regions of the graph.

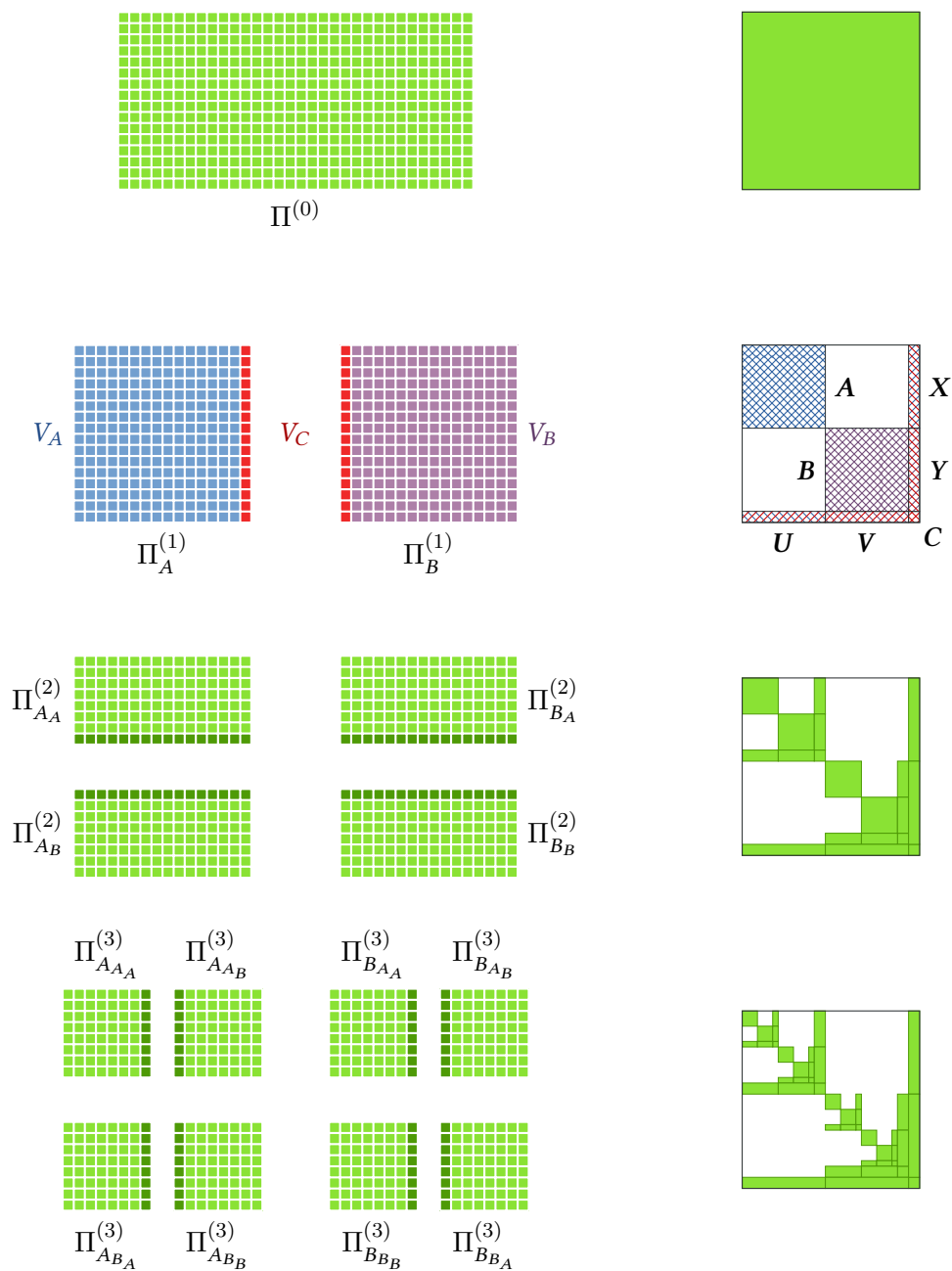


FIGURE 2.2: Nested dissection shown for a regular 2D-grid with (except for the margins) degree 8 (every “tile” in the grid represents a vertex). The 512 vertices of the 32×16 grid form a 512×512 adjacency matrix (sketched on the right). In each step l of the recursive process, the grid is split in two parts $\Pi_A^{(l)}$ and $\Pi_B^{(l)}$ which leads to the classification of the vertices into the sets V_A , V_B and V_C . The adjacency matrix is permuted such that it has the seven non-zero blocks defined in equation 2.63 and explained in table 2.1.

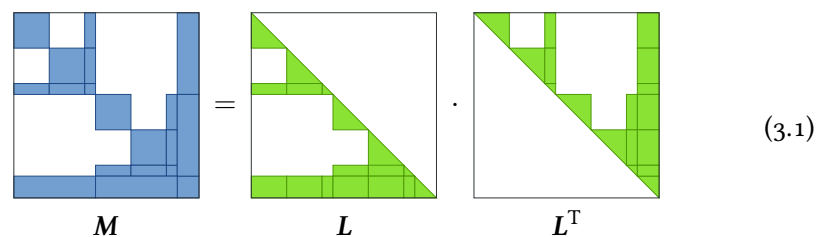
Chapter 3

Algorithmics

Let $n \in \mathbb{N}$ and $M \in \mathbb{R}^{n \times n}$ be a symmetric and positive definite H-matrix with a structure as if obtained by applying nested dissection (§ 2.4.1). Let further $k \in \mathbb{N}$ and $\mathbf{b}^{(1)}, \dots, \mathbf{b}^{(k)} \in \mathbb{R}^n$. We want to solve the linear systems $M\mathbf{x}^{(l)} = \mathbf{b}^{(l)}$ for $\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(k)} \in \mathbb{R}^n$ and $l \in \{1, \dots, k\}$.

We focus on symmetric and positive definite systems because they are a well-defined class of problems for which we know a unique LU factorization (that is, an LL^T factorization) without the need for permuting the coefficient matrix will always exist. This might seem to be a drastic assumption but it is justified by the fact that many real-world problems – for example those that originate from finite-element problems with symmetric interactions – are known to produce symmetric and positive definite systems¹. The algorithms we present are easily extended to the more general case that M is not symmetric and positive definite but only regular and does have an LU factorization without permuting, if this should be required².

In section 3.2 we present an algorithm to compute the block-wise decomposition $M = LL^T$ efficiently, exploiting the special structure of M . The obtained factorization will look like this:



$$M = L L^T \quad (3.1)$$

Once M is decomposed, a linear system is solved by performing a forward substitution on L and a backward substitution on L^T .

¹Also see footnote 24 on page 25.

²Note however, that the sheer fact that an LU factorization without permutation exists (in a mathematical sense) does not imply that it can be computed in a way that is numerically stable. In general, we should only assume this property for symmetric and positive definite matrices.

In section 3.3, we will show how this algorithm can be modified to compute a block-wise decomposition $M = LDL^T$ where D is a block-diagonal matrix, for which the inverse D^{-1} is known. This factorization looks like this:

$$M = L \cdot D \cdot L^T \quad (3.2)$$

Given this decomposition of M , a linear system is solved by a block-wise forward substitution on L followed by a multiplication with D^{-1} and finally a forward substitution on L^T . All three steps are mostly matrix-vector products which can be done very efficiently.

3.1 Notation

Let us denote the blocks of M with

$$M = M^{(0)} = \begin{pmatrix} A^{(1)} & \mathbf{0} & U^{(1)T} \\ \mathbf{0} & B^{(1)} & V^{(1)T} \\ U^{(1)} & V^{(1)} & C^{(1)} \end{pmatrix} \quad (3.3)$$

and the blocks of $A^{(1)}$ and $B^{(1)}$ with

$$A^{(1)} = \begin{pmatrix} A_A^{(2)} & \mathbf{0} & U_A^{(2)T} \\ \mathbf{0} & B_A^{(2)} & V_A^{(2)T} \\ U_A^{(2)} & V_A^{(2)} & C_A^{(2)} \end{pmatrix} \quad \text{and} \quad B^{(1)} = \begin{pmatrix} A_B^{(2)} & \mathbf{0} & U_B^{(2)T} \\ \mathbf{0} & B_B^{(2)} & V_B^{(2)T} \\ U_B^{(2)} & V_B^{(2)} & C_B^{(2)} \end{pmatrix}. \quad (3.4)$$

The blocks of $A_A^{(2)}$ would be named $A_{A_A}^{(3)}$, $B_{A_A}^{(3)}$, $C_{A_A}^{(3)}$ and so forth and analogously $A_{B_A}^{(3)}$, $B_{B_A}^{(3)}$, ... for $B_A^{(2)}$. In general, the child at position Z of block $W_R^{(l)}$ is named $Z_{W_R}^{(l+1)}$. Using English prose: the symbol on the base line denotes the position of the block in its parent block and the subscript names the parent block. The superscript merely counts the recursion depth. This syntax is redundant so we will optionally either drop the subscript part and only write $A^{(3)}$ to refer to any level-3 block at “position A” but don’t care where it is in the matrix or else write A_{A_A} if we want to be specific which block we mean. (Obviously, the superscript can be easily reconstructed by counting the number of recursive identifiers.) If we don’t even care about the recursion level, we will drop both, sub- and superscripts. Note that the blocks U , V and consequently their transposes as well as C never

have children. If we want to refer to the parent block of any block at level l but don't care where it is, we simply write $M^{(l-1)}$.

We don't make any special assumptions about the sizes of the sub-blocks at any level. In particular, we shall allow for n not to be a power of two, and the relative sizes of the blocks A , B and C may be different in each block. We do assume that A and B are "roughly the same size" and C be "considerably smaller", though. We also assume that the tree is "balanced", that is, at each level l , the sub-blocks $A^{(l+1)}$ and $B^{(l+1)}$ have the same number of children. This is not important for the correctness of the algorithms but otherwise some reasoning about their efficiency might not be appropriate.

We shall require the following from an algorithm to solve the systems:

- The total amount of work (FLOPs) should be as low as possible.
- The numeric errors should be kept below a fixed tolerance on the order of $\sqrt{\epsilon_{\text{float}}}$, where ϵ_{float} is the machine precision.
- The algorithm should scale well to massively parallel systems in the parallel random-access machine (PRAM) model. Ideally, it will keep a user-defined number of processing units busy with useful work for its entire execution time.

3.2 Block LL^T Decomposition of H-Matrices

The algorithm described in this section is based on previous work by Maurer and Wieners for a distributed computing environment [15] and private discussion with the author.

3.2.1 Decomposition

The structure of M can be exploited by using the block variant of the LU decomposition (§ 2.3.4). At level l , we first block the matrix $M^{(l)}$ to obtain

$$M^{(l)} = \left(\begin{array}{c|cc} A^{(l+1)} & \mathbf{0} & U^{(l+1)\text{T}} \\ \hline \mathbf{0} & B^{(l+1)} & V^{(l+1)\text{T}} \\ U^{(l+1)} & V^{(l+1)} & C^{(l+1)} \end{array} \right) = \begin{pmatrix} M_{11_A}^{(l)} & M_{21_A}^{(l)\text{T}} \\ M_{21_A}^{(l)} & M_{22_A}^{(l)} \end{pmatrix}. \quad (3.5)$$

We re-enter the algorithm recursively to factor

$$L_{11_A}^{(l)} L_{11_A}^{(l)\text{T}} = M_{11_A}^{(l)}, \quad (3.6)$$

solve the multiple triangular linear systems

$$L_{21_A}^{(l)} L_{11_A}^{(l)\text{T}} = M_{21_A}^{(l)} \quad (3.7)$$

and then compute

$$\tilde{\mathbf{M}}_A^{(l)} = \mathbf{M}_{22_A}^{(l)} - \mathbf{L}_{21_A}^{(l)} \mathbf{L}_{21_A}^{(l)T} \quad (3.8)$$

which gives us the following intermediate result.

$$\mathbf{M}^{(l)} = \begin{pmatrix} \mathbf{L}_{11_A}^{(l)} & \mathbf{0} \\ \mathbf{L}_{21_A}^{(l)} & \mathbf{1} \end{pmatrix} \begin{pmatrix} \mathbf{1} & \mathbf{0} \\ \mathbf{0} & \tilde{\mathbf{M}}_A^{(l)} \end{pmatrix} \begin{pmatrix} \mathbf{L}_{11_A}^{(l)T} & \mathbf{L}_{21_A}^{(l)T} \\ \mathbf{0} & \mathbf{1} \end{pmatrix} \quad (3.9)$$

The key observation is now that since the r topmost rows of $\mathbf{M}_{21_A}^{(l)}$ and hence the r leftmost columns of $\mathbf{M}_{12_A}^{(l)T}$ – where r is the size of $\mathbf{B}^{(l+1)}$ – are all zero,

$$\tilde{\mathbf{M}}_A^{(l)} = \begin{pmatrix} \mathbf{B}^{(l+1)} & \mathbf{V}^{(l+1)T} \\ \mathbf{V}^{(l+1)} & \mathbf{C}^{(l+1)} - \mathbf{\Delta}_A^{(l)} \end{pmatrix}. \quad (3.10)$$

Therefore, we can *independently* block $\mathbf{M}_{22_A}^{(l)}$ into

$$\left(\begin{array}{c|c} \mathbf{B}^{(l+1)} & \mathbf{V}^{(l+1)T} \\ \hline \mathbf{V}^{(l+1)} & \mathbf{C}^{(l+1)} \end{array} \right) = \begin{pmatrix} \mathbf{M}_{11_B}^{(l)} & \mathbf{M}_{21_B}^{(l)T} \\ \mathbf{M}_{21_B}^{(l)} & \mathbf{M}_{22_B}^{(l)} \end{pmatrix} \quad (3.11)$$

and *concurrently* re-enter our algorithm to analogously factor

$$\mathbf{L}_{11_B}^{(l)} \mathbf{L}_{11_B}^{(l)T} = \mathbf{M}_{11_B}^{(l)}, \quad (3.12)$$

solve the multiple triangular linear systems

$$\mathbf{L}_{21_B}^{(l)} \mathbf{L}_{11_B}^{(l)T} = \mathbf{M}_{21_B}^{(l)} \quad (3.13)$$

and compute

$$\tilde{\mathbf{M}}_B^{(l)} = \mathbf{M}_{22_B}^{(l)} - \mathbf{L}_{21_B}^{(l)} \mathbf{L}_{21_B}^{(l)T} = \mathbf{C}^{(l+1)} - \mathbf{\Delta}_B^{(l)}. \quad (3.14)$$

Combining our intermediate results, we get

$$\mathbf{M}^{(l)} = \begin{pmatrix} \mathbf{L}_{11_A}^{(l)} & \mathbf{0} \\ \mathbf{L}_{21_A}^{(l)} & \begin{array}{c|c} \mathbf{L}_{11_B}^{(l)} & \mathbf{0} \\ \hline \mathbf{L}_{21_B}^{(l)} & \mathbf{1} \end{array} \end{pmatrix} \begin{pmatrix} \mathbf{1} & \mathbf{0} \\ \mathbf{0} & \begin{array}{c|c} \mathbf{1} & \mathbf{0} \\ \hline \mathbf{0} & \tilde{\mathbf{M}}^{(l)} \end{array} \end{pmatrix} \begin{pmatrix} \mathbf{L}_{11_A}^{(l)T} & \mathbf{L}_{21_A}^{(l)T} \\ \mathbf{0} & \begin{array}{c|c} \mathbf{L}_{11_B}^{(l)T} & \mathbf{L}_{21_B}^{(l)T} \\ \hline \mathbf{0} & \mathbf{1} \end{array} \end{pmatrix} \quad (3.15)$$

with $\tilde{\mathbf{M}}^{(l)} := \mathbf{C}^{(l+1)} - (\mathbf{\Delta}_A^{(l)} + \mathbf{\Delta}_B^{(l)})$. Last, we factor

$$\mathbf{L}_{11}^{(l)} \mathbf{L}_{11}^{(l)T} = \mathbf{M}_{11}^{(l)} \quad (3.16)$$

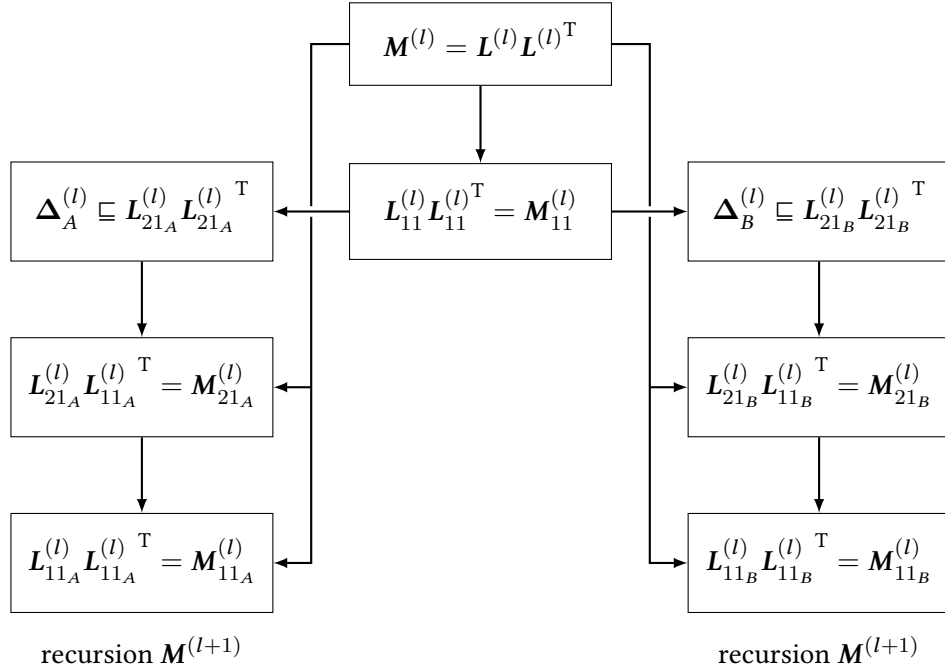


FIGURE 3.1: Data dependencies at level l for the block LL^T factorization of an H-matrix obtained via nested dissection. The arrows indicate dependencies. That is, “ $X \rightarrow Y$ ” means “ Y needs to be known for X to be computable”.

and finally get the complete factorization

$$\mathbf{M}^{(l)} = \underbrace{\begin{pmatrix} \mathbf{L}_{11A}^{(l)} & \mathbf{0} \\ \mathbf{L}_{21A}^{(l)} & \begin{matrix} \mathbf{L}_{11B}^{(l)} & \mathbf{0} \\ \mathbf{L}_{21B}^{(l)} & \mathbf{L}_{11}^{(l)} \end{matrix} \end{pmatrix}}_{:= \mathbf{L}^{(l)}} \underbrace{\begin{pmatrix} \mathbf{L}_{11A}^{(l)T} & \mathbf{L}_{21A}^{(l)T} \\ \mathbf{0} & \begin{matrix} \mathbf{L}_{11B}^{(l)T} & \mathbf{L}_{21B}^{(l)T} \\ \mathbf{0} & \mathbf{L}_{11}^{(l)T} \end{matrix} \end{pmatrix}}_{= \mathbf{L}^{(l)T}}. \quad (3.17)$$

Note that the last factorization is computed directly and does not re-enter the algorithm recursively. This is because we know nothing about the structure of $\mathbf{C}^{(l+1)}$ and therefore treat it as a single dense matrix. Recursion also ends if the matrices $\mathbf{A}^{(l+1)}$ and $\mathbf{B}^{(l+1)}$ do not have block structure.

3.2.2 Work-Flow

Let us now investigate the dependencies in the algorithm a bit further. Figure 3.1 illustrates the data dependencies between the various steps explained in the previous section.

At each level l of recursion and at each node $M^{(l)}$, the algorithm forks two independent dependency sub-trees for the decomposition of the sub-matrices $A^{(l+1)}$ and $B^{(l+1)}$ that are joined again via the final decomposition of the updated sub-matrix $C^{(l+1)} - (\Delta_A^{(l)} + \Delta_B^{(l)})$.

To formulate a simple yet effective recursive algorithm, we have it model the dependency tree. Each computation is performed as soon as all dependencies can be satisfied. As soon as the factorization of the top-left block (eq 3.6) is ready, we are able to solve the linear systems (eq 3.8) and compute the Schur complement (eq 3.7). Realizing that only leaf nodes (where the structure of the H-matrix does not further recurse) ever factorize, we can move all computation into the leaves. (Note that $C^{(l)}$ is always a leaf, at any level l .) Because the value of $M_{22}^{(l)}$ is never needed again, once $\tilde{M}^{(l)}$ is computed, we can update (overwrite) the matrix in-place instead of keeping the Schur complements in separate storage.

Algorithm 3.1 sketches out the basic framework for the recursive decomposition while algorithm 3.2 is the work-horse that actually decomposes the leaf nodes, writes the results into L and computes the Schur complements. The workflow is also illustrated more informally in figure 3.2.

3.2.3 Complexity

To be able to reason about the work that needs to be done, we need to introduce some parameters to describe the structure of the matrix. In equations 3.5 and 3.11 at level l , let the size of $M^{(l)}$ be $n^{(l)}$. For the blocks $M_{ijX}^{(l)}$ with $i, j \in \{1, 2\}$ and $X \in \{A, B\}$, let $n_{ijX}^{(l)}$ and $m_{ijX}^{(l)}$ denote their number of rows and columns respectively and

$$\rho_{ijX}^{(l)} := \frac{\text{NZC}\left(M_{ijX}^{(l)}\right)}{n_{ijX}^{(l)} m_{ijX}^{(l)}} \in [0, 1] \quad (3.18)$$

their respective sparsity factor where NZC denotes the number of entries in a matrix for which we don't know *a priori* (ie through the structure of the matrix) that they must be zero³. Since all work is done in the leaf nodes, we will drop the sub- and superscripts in the following discussion without ambiguity.

Table 3.1 summarizes the required FLOPs.

³To re-emphasize: If an entry inside a non-zero block just happens to have the value 0, it still counts as a non-zero entry since we'd first have to look at it to find out. For example, the 3×3 block diagonal matrix

$$X = \begin{pmatrix} Y & \mathbf{0} \\ \mathbf{0} & Z \end{pmatrix} \quad \text{with} \quad Y = \begin{pmatrix} 5 & 0 \\ 3 & 4 \end{pmatrix} \quad \text{and} \quad Z = (5)$$

has $\text{NZC}(X) = 5$ non-zero entries if the structure of Y is unspecified on input.

```

PROCEDURE NDBlockLLT
INPUT
     $M$  : HMatrix<REAL>
OUTPUT
     $L[1 \dots \text{size}(M)][1 \dots \text{size}(M)]$  : REAL
PRECONDITIONS
     $M$  is symmetric and positive definite
BEGIN
    Make a copy of  $M$  if it must not be destroyed during the process.
    CALL NDBlockLLTRecursive( $M$ ,  $M$ , 0,  $L$ )
END

PROCEDURE NDBlockLLTRecursive
INPUT
     $M$  : HMatrix<REAL>
     $M^{(l)}$  : HMatrix<REAL>
     $d^{(l)}$  : INTEGER
OUTPUT
     $L[1 \dots \text{size}(M)][1 \dots \text{size}(M)]$  : REAL
VARIABLES
     $A^{(l+1)}, B^{(l+1)}, C^{(l+1)}$  : HMatrix<REAL>
     $n_A^{(l+1)}, n_B^{(l+1)}$  : INTEGER
BEGIN
    IF the matrix  $M$  has children THEN
        Extract children  $A^{(l+1)}, B^{(l+1)}$  and  $C^{(l+1)}$  out of  $M^{(l)}$ .
         $n_A^{(l+1)} \leftarrow \text{size}(A^{(l+1)})$ 
         $n_B^{(l+1)} \leftarrow \text{size}(B^{(l+1)})$ 
        ASYNC BEGIN
            CALL NDBlockLLTRecursive( $M$ ,  $A^{(l+1)}$ ,  $d^{(l)}$ ,  $L$ )
            CALL NDBlockLLTRecursive( $M$ ,  $B^{(l+1)}$ ,  $d^{(l)} + n_A^{(l+1)}$ ,  $L$ )
        END
        CALL NDBlockLLTLeaf( $M$ ,  $C^{(l+1)}$ ,  $d^{(l)} + n_A^{(l+1)} + n_B^{(l+1)}$ ,  $L$ )
    ELSE
        CALL NDBlockLLTLeaf( $M$ ,  $M^{(l)}$ ,  $d^{(l)}$ ,  $L$ )
    FI
END

```

ALGORITHM 3.1: The procedure NDBLOCKLLTRecursive is the skeleton for the recursive block LL^T decomposition of an H-matrix with nested dissection structure. NDBlockLLT merely provides a clean interface that hides the implementation details from the user. The work horse NDBlockLLTLeaf is shown in algorithm 3.2.

```

PROCEDURE NDBlockLLTLeaf
INPUT
   $M$  : HMatrix<REAL>
   $M_{11}^{(l)}$  [1 ...  $n_{11}^{(l)}$ ][1 ...  $n_{11}^{(l)}$ ] : REAL
   $d^{(l)}$  : INTEGER
OUTPUT
   $L$  [1 ... size( $M$ )][1 ... size( $M$ )] : REAL
PRECONDITIONS
   $M$  is symmetric and positive definite
   $M_{11}^{(l)}$  is a main diagonal block of  $M$ 
VARIABLES
   $L_{11}^{(l)}$  [1 ...  $n_{11}^{(l)}$ ][1 ...  $n_{11}^{(l)}$ ] : REAL
   $M_{21}^{(l)}$  [1 ... size( $M$ ) -  $d$  -  $n_{11}^{(l)}$ ][1 ...  $n_{11}^{(l)}$ ] : REAL
   $L_{21}^{(l)}$  [1 ... size( $M$ ) -  $d$  -  $n_{11}^{(l)}$ ][1 ...  $n_{11}^{(l)}$ ] : REAL
   $M_{22}^{(l)}$  [1 ... size( $M$ ) -  $d$  -  $n_{11}^{(l)}$ ][1 ... size( $M$ ) -  $d$  -  $n_{11}^{(l)}$ ] : REAL
BEGIN
  ;; Create views for the blocks; these shall read and write through.
   $L_{11}^{(l)} \leftarrow L[d + 1 : d + n_{11}^{(l)}][d + 1 : d + n_{11}^{(l)}]$ 
   $M_{21}^{(l)} \leftarrow M[d + n_{11}^{(l)} + 1 : \text{size}(M)][d + 1 : d + n_{11}^{(l)}]$ 
   $L_{21}^{(l)} \leftarrow L[d + n_{11}^{(l)} + 1 : \text{size}(M)][d + 1 : d + n_{11}^{(l)}]$ 
   $M_{22}^{(l)} \leftarrow M[d + n_{11}^{(l)} + 1 : \text{size}(M)][d + n_{11}^{(l)} + 1 : \text{size}(M)]$ 
  CALL Cholesky( $M_{11}^{(l)}, L_{11}^{(l)}$ )
  CALL BackwardSubstitution( $L_{11}^{(l)T}, M_{21}^{(l)}, L_{21}^{(l)}$ )
   $M_{22}^{(l)} \leftarrow M_{22}^{(l)} - L_{21}^{(l)} L_{21}^{(l)T}$ 
END

```

ALGORITHM 3.2: Implementation of the NDBLOCKLLTLEAF routine called from algorithm 3.1. It decomposes a single dense block $M_{11}^{(l)}$ on the main diagonal of M , writes the respective results $L_{11}^{(l)}$ and $L_{21}^{(l)}$ into L and overwrites $M_{22}^{(l)}$ to become the Schur complement $\tilde{M}^{(l)}$. Note that this implementation assumes atomic arithmetic and sequential consistency. Please see the text (§ 3.2.4.1) for a detailed discussion and how the algorithm can be adapted to mitigate race conditions.

Task	Operation	Required FLOPs
$L_{11}L_{11}^T = M_{11}$	factorization	$\frac{1}{6}n_{11}^3$
$L_{21}L_{11}^T = M_{21}$	backward substitution	$\frac{1}{2}\rho_{21}n_{21}n_{11}^2$
$\Delta \sqsubseteq L_{21}L_{21}^T$	multiplication ($\omega = 3$)	$\rho_{21}^2n_{21}^2n_{11}$

TABLE 3.1: Work to be done in leaf nodes. Note that $n_{12} = n_{11} = m_{11} = n_{21}$ and $m_{12} = m_{22} = n_{22} = n_{21}$. Also note that $\rho_{11} = 1$ and that all rows in M_{21} are either all-zero or all-non-zero, hence, $\rho_{21}n_{21}$ is the number of non-zero rows in M_{21} .

3.2.4 Parallelism

3.2.4.1 Synchronization

Note how the recursive formulation of the algorithm elegantly models the data dependencies. As usual, we have not cluttered our algorithms (algo 3.1 and 3.2) with synchronization mechanisms but assumed atomic arithmetic and sequential consistency. This is of course an oversimplification if it comes to real hardware. Synchronization is expensive and partially destroys concurrency so it might be worthwhile to examine the possible race conditions and discuss modifications to the algorithm that mitigate them without the need for locking.

The first and pleasuring observation is that there are no possible race conditions on L . Each entry in L is computed and written to exactly once at a well-defined point in the algorithm. Therefore, the results can be written to them just as shown in the listings.

Unfortunately, there *is* a race condition on the updates for the Schur complement as we subtract $\Delta_A^{(l)}$ and $\Delta_B^{(l)}$ and then later also $\Delta_C^{(l)}$ – which actually is either $\Delta_A^{(l-1)}$ or $\Delta_B^{(l-1)}$ of the caller, we just don’t happen to know – from $M_{22}^{(l)}$.

The first race occurs between the children of a node when when the two asynchronous calls to NDBLOCKLLTRECURSIVE for factoring $A^{(l+1)}$ and $B^{(l+1)}$ write simultaneously to apply their respective updates. The second race takes place between the node and its sister node as it writes the update obtained from factoring $C^{(l+1)}$. Fortunately, a closer look reveals that not all of the update participates in the race. While the code in NDBLOCKLLTLEAF cannot possibly avoid the race, the caller in NDBLOCKLLTRECURSIVE knows that after block $A^{(l+1)}$ and $B^{(l+1)}$ have been factored, not only is it safe but actually required to write the updates – one after each other – to $C^{(l+1)}$ or the algorithm could not continue. Likewise, it is equally safe to write to regions of M that are located on the same column or on the same row as an entry in $C^{(l+1)}$. Put another way, the only region that is subject to races is the “south east” corner below $C^{(l+1)}$.

This suggests a slight re-design of algorithm 3.1 and 3.2. There is no way

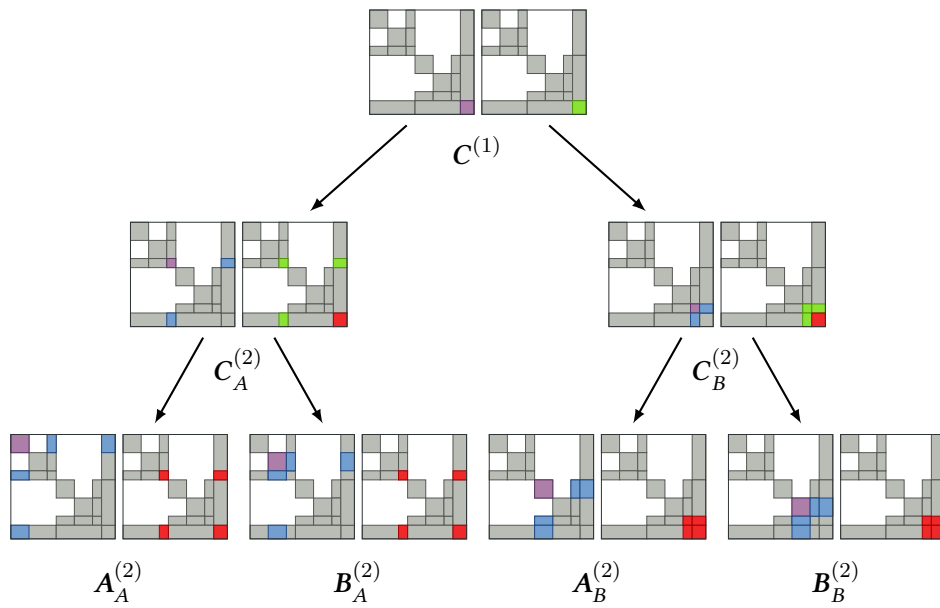


FIGURE 3.2: Call tree for the block LL^T decomposition of an H-matrix with nested dissection structure. The two matrices shown at each node indicate the affected regions that have read (left) and write (right) dependencies. Printed below the matrices is the block that is factorized at the current node. The colors mark the following blocks: “ \blacksquare ” for the block $M_{11}^{(l)}$ that is factorized, “ \blacksquare ” for the block $M_{21}^{(l)}$ that is substituted for and its respective transpose, “ \blacksquare ” for the parts of $M_{22}^{(l)}$ that have pending updates (passed up by child functions) and may safely be written to and “ \blacksquare ” for the regions that are subject to data races and must not be written to. Instead, their updates have to be propagated up the call tree.

to write safely from within `NDBLOCKLLTLEAF` so instead of having it write the update directly, we introduce an additional output parameter that references some temporary storage where it may place the update. We also add such a parameter to `NDBLOCKLLTRECURSIVE` but this time, it only needs to cover the region south east of $C^{(l)}$. The other parts of the updates from factoring $A^{(l)}$ and $B^{(l)}$ we subtract from M . The other parts, of $\Delta_A^{(l)}$ and $\Delta_B^{(l)}$ we add together with $\Delta_C^{(l)}$ that is passed up to our caller and then release the temporary storage allocated for the children. This way, the unhandled updates propagate up the call tree until the root node writes the last update and factorizes $C^{(1)}$ at which point there is no more update and the algorithm has completed its work. Figure 3.2 visualizes these data dependencies on the call tree.

This modified algorithm allows for a fully lock-free implementation even on real hardware.

3.2.4.2 Parallel Efficiency

The results from section 3.2.4.1 are pleasing. However, let us consider what happens if the matrix is decomposed on a shared-memory machine with $p \in \mathbb{N}$ independent processing units. For simplicity – and also because that’s what most hardware actually provides – let’s assume that p is a power of two. As long as the recursion is operating at a level $l \in \mathbb{N}$ such that $2^l \geq p$, each unit can process one branch of recursion at a time and all hardware will be made optimal use of. Since the descent in the tree does not require any actual work to be done, it is negligible and computation starts almost instantly at the leaf-level. The expensive part is the ascent that carries up the dependencies to the callers. However, as the recursion returns up to higher levels, the number of concurrently processed nodes at some point will drop below the number of available cores. It now depends on how well the algorithms used for the work that has to be done at every node (mostly dense factorization, substituting and matrix multiplication) are able to utilize the spare cores. Note that it are exactly those nodes very near to the root of the tree that have the largest blocks and therefore take the longest to process.

For the matrix-matrix products, this is not a problem as it can be done fully parallel. However, factorization and substitution both have a critical path length on the order of $\Omega(n^{(l)})$. Therefore, in a massively parallel environment, the algorithm might not be able to make efficient use of all processing units.

3.2.5 Solving

Once the decomposition is computed, any number of linear systems can be solved by performing a forward substitution on L followed by a backward substitution on L^T . However, for the procedure doing this to exploit the special structure of L , the substitution algorithm has to be reformulated for H-matrices.

Let us have a look at the structure of L . Every time a main diagonal block M_{11} is factored, a triangular block L_{11} is inserted on the main diagonal and a “column”

L_{21} is inserted below it. Note that L_{21} is again hierarchical containing blocks of non-zero rows. Using the picture from equation 3.1, we can visualize this as

$$\begin{matrix} \text{[Matrix } L \text{ with green blocks]} \\ \cdot \\ \mathbf{y}^{(l)} \end{matrix} = \mathbf{b}^{(l)} \quad \text{and} \quad \begin{matrix} \text{[Matrix } L^T \text{ with green blocks]} \\ \cdot \\ \mathbf{x}^{(l)} \end{matrix} = \mathbf{y}^{(l)} \quad (3.19)$$

for $l \in \{1, \dots, k\}$. (We will drop the superscripts for the following discussion since they don't matter as each system is completely independent of the others.)

Carefully looking at the pictures suggests that we solve the equations piece-wise. For each triangular main diagonal block L_{11} in L , let $n_{11} := \text{size}(L_{11})$ and $d_{11} \in \{1, \dots, n\}$ such that $L_{11} = L[d_{11} + 1 : d_{11} + n_{11}][d_{11} + 1 : d_{11} + n_{11}]$. Then the ‘‘piece’’ $\mathbf{y}[d_{11} + 1 : d_{11} + n_{11}]$ can be computed by an ordinary forward substitution to solve

$$\mathbf{b}[d_{11} + 1 : d_{11} + n_{11}] = L_{11} \hat{\mathbf{y}}_{11} \quad (3.20)$$

where

$$\hat{\mathbf{y}}_{11} := \mathbf{b}[d_{11} + 1 : d_{11} + n_{11}] - L[d_{11} + 1 : d_{11} + n_{11}][1 : d_{11}] \mathbf{y}[1 : d_{11}] \quad (3.21)$$

The cost for this is $\frac{1}{2}n_{11}^2$ FLOPs for the substitution and $\rho_{11}n_{11}d_{11}$ FLOPs for the matrix-vector products where ρ_{11} is the sparsity factor of $L[d_{11} + 1 : d_{11} + n_{11}][1 : d_{11}]$ which might be 0. The critical path is determined by the substitutions, which means that it remains on the order of $\Omega(n)$, even though the matrix-vector products can be computed fully parallel.

The procedure for the backward substitution to solve $L^T \mathbf{x} = \mathbf{y}$ is analogous (and hence also has the same cost and critical path).

3.3 Block LDL^T Decomposition of H-Matrices

In this section, we present an application of matrix inversion to a modification of the algorithm described in the previous section to yield a block LDU decomposition of M (or a block LDL^T decomposition, that is).

The required changes are relatively small so we will refer to the detailed discussion in the previous section except for the parts that are different. In particular, the recursive framework of the algorithm stays the same. Only at the leaf nodes, we do not factor $L_{11}^{(l)} L_{11}^{(l)T} = M_{11}^{(l)}$ but instead compute the inverse $M_{11}^{(l)-1}$. Once this is done, instead of doing substitutions, we compute

$$L_{21}^{(l)} = M_{21}^{(l)} M_{11}^{(l)-1} \quad (3.22)$$

and

$$\tilde{\mathbf{M}}^{(l)} = \mathbf{M}_{22}^{(l)} - \mathbf{M}_{21}^{(l)} \mathbf{M}_{11}^{(l)-1} \mathbf{M}_{21}^{(l)\top} = \mathbf{L}_{21}^{(l)} \mathbf{M}_{21}^{(l)\top}. \quad (3.23)$$

Of course, the inverse is not thrown away but kept for when we'll need it later to actually solve a linear system.

Without any further ado, let us present in algorithm 3.3 a modification of algorithm 3.2 that implements the above. The only change required in the code that calls it (cf algo 3.1) is the introduction of the additional output parameter for \mathbf{D}^{-1} . Since this is a rather trivial change, we do not duplicate all of the rest of the code for it.

With the block \mathbf{LDL}^\top decomposition at hand, we can solve the linear systems

$$\mathbf{M}\mathbf{x}^{(l)} = \mathbf{b}^{(l)}$$

equivalent to

$$\mathbf{LDL}^\top \mathbf{x}^{(l)} = \mathbf{b}^{(l)}$$

for $l \in \{1, \dots, k\}$ in three steps.

First, we solve

$$\mathbf{L}\mathbf{z}^{(l)} = \mathbf{b}^{(l)} \quad (3.24)$$

for $\mathbf{z}^{(l)} \in \mathbb{R}^n$ by means of a block forward substitution. Next, we solve

$$\mathbf{D}\mathbf{y}^{(l)} = \mathbf{z}^{(l)} \quad (3.25)$$

for $\mathbf{y}^{(l)} \in \mathbb{R}^n$ by using $\mathbf{y}^{(l)} = \mathbf{z}^{(l)} \mathbf{D}^{-1}$ (note that we already have the inverse \mathbf{D}^{-1} handy). Finally, we solve

$$\mathbf{L}^\top \mathbf{x}^{(l)} = \mathbf{y}^{(l)} \quad (3.26)$$

to obtain the wanted $\mathbf{x}^{(l)} \in \mathbb{R}^n$ via a backward substitution.

Let us view equations 3.24 to 3.26 graphically. First, the solution for

$$\mathbf{D} \cdot \mathbf{y}^{(l)} = \mathbf{z}^{(l)} \quad (3.27)$$

is obtained by a straight-forward matrix-vector multiplication with the inverse \mathbf{D}^{-1} that has the same block diagonal structure as \mathbf{D} . The substitutions

$$\mathbf{L} \cdot \mathbf{z}^{(l)} = \mathbf{b}^{(l)} \quad \text{and} \quad \mathbf{L}^\top \cdot \mathbf{x}^{(l)} = \mathbf{y}^{(l)} \quad (3.28)$$

```

PROCEDURE NDBlockLDLTLeaf
INPUT
   $M$  : HMatrix<REAL>
   $M_{11}^{(l)}$  [1 ...  $n_{11}^{(l)}$ ][1 ...  $n_{11}^{(l)}$ ] : REAL
   $d^{(l)}$  : INTEGER
OUTPUT
   $L$  [1 ... size( $M$ )][1 ... size( $M$ )] : REAL
   $D^{-1}$  [1 ... size( $M$ )][1 ... size( $M$ )] : REAL
PRECONDITIONS
   $M$  is symmetric and positive definite
   $M_{11}^{(l)}$  is a main diagonal block of  $M$ 
VARIABLES
   $D_{11}^{(l)-1}$  [1 ...  $n_{11}^{(l)}$ ][1 ...  $n_{11}^{(l)}$ ] : REAL
   $M_{21}^{(l)}$  [1 ... size( $M$ ) -  $d$  -  $n_{11}^{(l)}$ ][1 ...  $n_{11}^{(l)}$ ] : REAL
   $L_{21}^{(l)}$  [1 ... size( $M$ ) -  $d$  -  $n_{11}^{(l)}$ ][1 ...  $n_{11}^{(l)}$ ] : REAL
   $M_{22}^{(l)}$  [1 ... size( $M$ ) -  $d$  -  $n_{11}^{(l)}$ ][1 ... size( $M$ ) -  $d$  -  $n_{11}^{(l)}$ ] : REAL
BEGIN
  ;; Create views for the blocks; these shall read and write through.
   $D_{11}^{(l)-1} \leftarrow D^{-1}[d + 1 : d + n_{11}^{(l)}][d + 1 : d + n_{11}^{(l)}]$ 
   $M_{21}^{(l)} \leftarrow M[d + n_{11}^{(l)} + 1 : \text{size}(M)][d + 1 : d + n_{11}^{(l)}]$ 
   $L_{21}^{(l)} \leftarrow L[d + n_{11}^{(l)} + 1 : \text{size}(M)][d + 1 : d + n_{11}^{(l)}]$ 
   $M_{22}^{(l)} \leftarrow M[d + n_{11}^{(l)} + 1 : \text{size}(M)][d + n_{11}^{(l)} + 1 : \text{size}(M)]$ 
  CALL InvertSPD( $M_{11}^{(l)}$ ,  $D_{11}^{(l)-1}$ )
   $L_{21}^{(l)} \leftarrow M_{21}^{(l)} D_{11}^{(l)-1}$ 
   $M_{22}^{(l)} \leftarrow M_{22}^{(l)} - L_{21}^{(l)} L_{21}^{(l)T}$ 
END

```

ALGORITHM 3.3: Modification of the NDBLOCKLLTLEAF routine called from algorithm 3.1. It inverts a single dense block $M_{11}^{(l)}$ on the main diagonal of M , and writes the results $M_{11}^{(l)-1}$ and $L_{21}^{(l)}$ into D^{-1} and L respectively. $M_{22}^{(l)}$ is overwritten to become the Schur complement $\tilde{M}^{(l)}$. Note that this implementation assumes atomic arithmetic and sequential consistency. The discussion in section 3.2.4.1 about race conditions applies equally.

Task	Operation	Required FLOPs
M_{11}^{-1}	symmetric inversion	$\frac{1}{2}n_{11}^3$
$L_{21} = M_{21}M_{11}^{-1}$	multiplication ($\omega = 3$)	$\rho_{21}n_{21}n_{11}^2$
$\Delta \sqsubseteq L_{21}L_{21}^T$	multiplication ($\omega = 3$)	$\rho_{21}^2n_{21}^2n_{11}$

TABLE 3.2: Work to be done in leaf nodes for the block LDL^T decomposition. Compared with the block LL^T factorization (tab 3.1), the inversion is three times more expensive than the Cholesky factorization and the matrix-matrix product to compute L_{21} costs twice the number of FLOPs for the substitution. The computation of the update Δ is identical.

are also “easy” in the sense that the corresponding linear systems fall apart into sets of equations such that all equations in a set can be solved independently of each other. Therefore, they are computed like in equations 3.20 and 3.21 except that since $L_{11} = \mathbf{1}$, no substitution is needed and $\mathbf{b}[d_{11} + 1 : d_{11} + n_{11}] = \hat{\mathbf{y}}$.

3.3.1 Complexity

Let us now compare the complexity and efficiency of the block LDL^T decomposition with that of the block LL^T decomposition. Table 3.2 details the cost for the decomposition. Compared to the block LL^T decomposition, the additional work is significant. When solving, the only difference is that where a forward and backward substitution for the triangular main diagonal blocks on L – accounting for $\frac{1}{2}n_{11}^2$ FLOPs each – is performed in the block LL^T case, a single equally expensive (n_{11}^2 FLOPs) matrix-vector product is computed in case of the block LDL^T decomposition. Since the multiplication is “simpler”, we expect it to be computable with less overhead.

3.3.2 Parallelism

The distinguishing feature of the presented block LDL^T decomposition is that if an appropriate matrix inversion algorithm – NeSt (§ 2.3.5.4), for example – is plugged, the complete decomposition as well as the procedure for solving is parallelized very well.

To see this for the decomposition, recall that the height of the dependency tree (fig 3.2) is logarithmic in the matrix size. At each node in the tree, only matrix inversion and multiplication is performed, both of which can be done fully parallel. This yields an overall critical path on the order of $\Omega(\log(n)T_{\text{inv}}(n))$ for the entire algorithm where T_{inv} is the critical path for matrix inversion and can also be made logarithmic.

In case of solving, the number of blocks in L that define sets of independent

equations, is identical to the number of main diagonal blocks in M . Unfortunately, this is not logarithmic but still somehow linear in n . However, if we assume that n_{\min} is the size of the smallest such block, then we will have reduced the critical path by at least a factor of n_{\min} compared to block LL^T decomposition. Usually, the smallest block will be chosen “quite large” in order for library functions to operate efficiently. Therefore, the improvement might be quite considerable.

Chapter 4

Implementation

We have implemented the algorithms for block LL^T (§ 3.2) and block LDL^T (§ 3.3) decomposition for a shared memory system. The block LDL^T decomposition uses the NeSt algorithm (§ 2.3.5.4) for matrix inversion.

In this chapter we present the basic concepts of our implementation. The implementation is Free Software. The source code may be obtained from the author.¹ The package is a research project and not industry-strength software. However, we did try to follow best practices for programming high-quality software.

4.1 Technologies

4.1.1 Programming Languages

We have implemented our algorithms in C++. This choice was made because of the excellent combination of high performance and abstractions the language provides, especially since the C++11 revision of the standard. Our code makes heavy use of C++11 features.

The decision for using C++ was further influenced by the fact that Sanders, Speck, and Steffen have implemented their NeSt algorithm in C++ too so we could re-use it more easily.

In the remainder of this chapter, we will assume that the reader has a fairly intermediate level of knowledge about the C++ programming language, in particular, the features introduced in C++11. In case of doubt, please refer to Stroustrup [22].

¹<http://www.klammler.eu/bsc/> – Unfortunately, we cannot release the source code of the NeSt algorithm due to its unclear copyright status. Please try writing to its authors to obtain a copy of NeSt’s source code for private use. You can compile and run our software without NeSt but then a less parallel library function for matrix inversion will be used. While it is possible and legal (but in no way required) to link our software to Intel’s *Math Kernel Library*, doing so makes the resulting program non-free software.

4.1.2 Libraries

Routines for dense linear algebra operations are among the best understood and most heavily optimized pieces of software. It follows that we wanted to delegate as much work as possible to off-the-shelf libraries and not implement our own versions of standard algorithms. Therefore, we have considered a number of libraries to base our work upon. Our requirements were that the library shall be

- *fast* — so we can compare our results to the current industry standard,
- *concurrent* — because we are interested how algorithms scale in a highly parallel context,
- *flexible* — meaning that ideally we can program against a stable interface and plug different kernels without having to change our code,
- *elegant* — providing clean abstractions and powerful high-level syntax leveraging C++’ operator overloading and template meta-programming capabilities,
- *mature* — so we can be reasonably sure it works correctly on major platforms and will still do so in the future and
- *free software* — because we did not want our research to be locked in by a single vendor and wanted to give everyone the possibility to reproduce our results without the need to buy a certain proprietary product first. We also consider it important for a research project (and other projects as well) to be able to study the internals of the mission-critical third-party components.

4.1.2.1 BLAS & LAPACK

The low-level *Basic Linear Algebra Subprograms* (BLAS) [2] with the more high-level *Linear Algebra Package* (LAPACK) [13] built on top are a collection of Fortran library interfaces that have for many decades built the foundation of high-performance numeric computing. Fairly standard bindings for the C programming language (CBLAS)² exist so the libraries can be used from C and C++ code as well.

²The `cblas.h` header file as well as a thin compatibility layer written in C is available as free software from the Netlib repository (http://www.netlib.org/blas/#_cblas). The wrapper library can be compiled into a small archive that can be statically linked into the application. It is then possible to link an application written in C or C++ with a BLAS library just as if the application were natively written in Fortran in the first place. (The “as if” is to be taken literally. It is required to link with the Fortran run-time and on some systems, a Fortran-style “dummy main” function (that does nothing) must be included in the C program. The GNU Autotools can greatly help with these portability issues.)

There is a mostly unoptimized reference implementation available as free software from the *Netlib* repository³ as well as a number of highly optimized implementations as both, free and proprietary software. Since the application programming interface (API) has become a de facto standard, these implementations may be exchanged without the need to edit (or even re-compile) the application's code. On the machines we have used to test our code, a copy of Intel's proprietary *Math Kernel Library* (MKL) [14] was installed. This library is famous for its speed but being non-free, there is no way to figure out how exactly its internals are working. Some BLAS implementations are multi-threaded and others are not. Except for execution time, this fact is transparent to the user. Intel's MKL comes in two flavors, sequential and multi-threaded.

On the other hand, the API is a relic of the Fortran area and barely provides any abstractions beyond arrays, numbers and functions. This design is also incapable to make use of compile-time optimizations that are enabled by the use of modern template meta-programming. Instead, the programmer who is using the BLAS has to do what could otherwise be done by a high-level template library and an optimizing compiler.

Let $n, k, m \in \mathbb{N}$, $A \in \mathbb{R}^{n \times k}$, $B \in \mathbb{R}^{m \times k}$, $C \in \mathbb{R}^{n \times m}$ and $\alpha, \beta \in \mathbb{R}$. To compute the expression

$$\alpha AB^T + \beta C \tag{4.1}$$

and store the result in C , overwriting its previous contents, one would use the CBLAS interface like shown in listing 4.1. While the intent of the program will be obvious to experienced BLAS programmers, it is not immediately clear simply from reading the source code. What is probably most disturbing is that the programmer has to carry around the dimensions in separate variables. This is tedious and error prone. Of course, one could wrap a pointer to the matrix data together with its dimensions into a C++ class and provide overloaded operators that call into the BLAS. We did consider this option but refrained from using it since for any but the most trivial expressions, it would either have to introduce significant abstraction overhead or else be very hard to implement. Most importantly, there already exist libraries that provide better abstractions than we could have implemented in an ad hoc manner.

4.1.2.2 Boost uBLAS

Boost [3] is a collection of high-quality peer-reviewed free software C++ libraries that are sometimes viewed as a sandbox for contributions that some day will eventually become part of a future C++ standard. *uBLAS* [23] is part of Boost's numeric library. From its website⁴:

³<http://www.netlib.org/>

⁴http://beta.boost.org/doc/libs/1_56_0/libs/numeric/ublas/doc/index.htm


```

#include <cstddef>
#include <cblas.h>

void
example(const std::size_t n,
        const std::size_t k,
        const std::size_t m,
        const double * a_ptr,
        const double * b_ptr,
        double * c_ptr,
        const double alpha,
        const double beta)
{
    cblas_dgemm(CblasColMajor, // data layout
               CblasNoTrans,  // don't transpose A
               CblasTrans,    // do transpose B
               n,              // rows(A) = rows(C)
               m,              // cols(BT) = cols(C)
               k,              // cols(A) = rows(BT)
               alpha,          // α
               a_ptr,          // pointer to the data of A
               n,              // stride for A (special case)
               b_ptr,          // pointer to the data of B
               m,              // stride for B (special case)
               beta,           // β
               c_ptr,          // pointer to the data of C
               n);             // stride for C (special case)
}

```

LISTING 4.1: Example code demonstrating the use of the CBLAS interface to compute $\alpha\mathbf{AB}^T + \beta\mathbf{C}$ and store the result in \mathbf{C} . The code makes some assumptions that do not hold in general. In particular, the data layout need not be “column major” and the strides (called LDX for parameter X in the BLAS jargon) need not match the dimensions of the matrix, therefore allowing to slice out sub-matrices.

uBLAS is a C++ template class library that provides BLAS level 1, 2, 3 functionality for dense, packed and sparse matrices. The design and implementation unify mathematical notation via operator overloading and efficient code generation via expression templates.

Using Boost uBLAS, the expression from equation 4.1 can be computed as shown in listing 4.2. While this is not particularly elegant, the intent of the program is pretty clear even to the unacquainted. Most importantly, it gets rid of the tedious bookkeeping required when using the CBLAS interface.

Boost uBLAS can use its own C++ algorithms based on expression templates (this is the default) or be bound to external BLAS libraries. In particular, Intel distributes a header file to bind computation of dense matrix-matrix products to their proprietary MKL [11]. Unfortunately, this binding is only very rudimentary and might require intrusive changes to the code. For example, the expression in listing 4.2 has to be re-written to use a temporary. We did not succeed at binding Boost uBLAS to other BLAS kernels but there are reports on the internet that suggest other people did.

uBLAS itself is not multi-threaded so the only way to gain parallelism is to bind it to the multi-threaded MKL.

```
#include <boost/numeric/ublas/matrix.hpp>

void
example(const boost::numeric::ublas::matrix<double>& a,
        const boost::numeric::ublas::matrix<double>& b,
        boost::numeric::ublas::matrix<double>& c,
        const double alpha,
        const double beta)
{
    c = alpha * prod(a, trans(b)) + beta * c;
}
```

LISTING 4.2: Example code demonstrating the use of the Boost uBLAS template library to compute $\alpha AB^T + \beta C$ and store the result in C . In the above code, `prod` and `trans` bind to `boost::numeric::ublas::prod` and `boost::numeric::ublas::trans` respectively that are found via ADL.

4.1.2.3 TNT

The *Template Numerical Toolkit* (TNT) [18] is a free software C++ numerical library developed by Pozo from the National Institute of Standards and Technology (NIST). From the TNT website⁵:

⁵<http://math.nist.gov/tnt/>

[TNT] is a collection of interfaces and reference implementations of numerical objects useful for scientific computing in C++. The toolkit defines interfaces for basic data structures, such as multidimensional arrays and sparse matrices, commonly used in numerical applications. The goal of this package is to provide reusable software components that address many of the portability and maintenance problems with C++ codes.

TNT provides a distinction between *interfaces* and *implementations* of TNT components. For example, there is a TNT interface for two-dimensional arrays which describes how individual elements are accessed and how certain information, such as the array dimensions, can be used in algorithms; however, there can be *several* implementations of such an interface: one that uses expression templates, or one that uses BLAS kernels, or another that is instrumented to provide debugging information. By specifying only the *interface*, applications codes may utilize such algorithms, while giving library developers the greatest flexibility in employing optimization or portability strategies.

While, according to this description, TNT seems to provide just what we need, we could not figure out how to make efficient use of the library. We have managed to compute simple expressions using TNT but this was neither fast nor convenient. We have not been able to bind TNT to an external BLAS library. Since there didn't seem to be a consistent and up-to-date reference manual, reading the source code was often the only way to learn about the library. This was further complicated by the fact that there are two versions (TNT 1.2.6 and TNT 3.0.12) with different APIs available at the NIST website. Considering these circumstances, we have quickly abandoned the option to use TNT for our work.

Listing 4.3 shows our best attempt to implement equation 4.1 inside the TNT. It is inferior compared to all other solutions in that it creates temporary objects for intermediate results, instead of computing the entire expression in one flush.

4.1.2.4 Eigen

The last library we have considered was *Eigen* [10]; a stand-alone free software linear algebra library. Its website ⁶ says:

Eigen is a C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms.

Eigen uses expression templates to not only minimize abstraction overhead but actually leverage compile-time optimizations that are not possible using ordinary function calls. This makes the C++ code very idiomatic without losing performance. The example from equation 4.1 is implemented with Eigen as shown

⁶<http://eigen.tuxfamily.org/>

```

#include <tnt/tnt.h>

void
example(const TNT::Matrix<double>& a,
        const TNT::Matrix<double>& b,
        TNT::Matrix<double>& c,
        const double alpha,
        const double beta)
{
    c = mult(alpha, mult(a, transpose(b))) + mult(beta, c);
}

```

LISTING 4.3: Example code demonstrating the use of the Eigen template library to compute $\alpha \mathbf{AB}^T + \beta \mathbf{C}$ and store the result in \mathbf{C} . As it stands, this code creates up to four temporary objects unless the compiler is able to optimize some of them away. `mult` and `transpose` bind to `TNT::mult` and `TNT::transpose` via ADL.

in listing 4.4. It is almost identical to the mathematical notation. In fact, Eigen provides even much more powerful features to write complex mathematical expressions in a very elegant way.

Eigen can be parallelized using OpenMP. All that needs to be done for this is enabling OpenMP in the compiler⁷. It is possible to bind Eigen to the MKL simply by defining a preprocessor symbol. However, as of this writing, it is not possible to bind it to other BLAS kernels.

Eigen is under active development by an open volunteer group of free software hackers, many with academic backgrounds. It has an active project mailing list and our questions and patches were promptly dealt with.

```

#include <Eigen/Dense>

void
example(const Eigen::MatrixXd& a,
        const Eigen::MatrixXd& b,
        Eigen::MatrixXd& c,
        const double alpha,
        const double beta)
{
    c = alpha * a * b.transpose() + beta * c;
}

```

LISTING 4.4: Example code demonstrating the use of the Eigen template library to compute $\alpha \mathbf{AB}^T + \beta \mathbf{C}$ and store the result in \mathbf{C} . `MatrixXd` is a convenience typedef for dynamic-size matrices with values of type `double`.

⁷For the GNU C++ compiler, the respective flag would be `-fopenmp`.

Library / Interface	Speed	Concurrency	Flexibility	Elegance	Maturity	Free Software
CBLAS	✓ ^a	✓ ^a	✓	✗	✓	✓ ^a
Boost uBLAS	✓ ^b	✗ ^b	✓ ^c	✓	✓	✓ ^d
TNT	✗ ^e	✗ ^e	✗ ^e	✗	✗	✓
Eigen	✓	✓	✓ ^c	✓	✓	✓ ^d

^a depends on the BLAS library that is linked to

^b good if compiled with MKL bindings

^c can be bound to the MKL but not to any BLAS implementation in general

^d unless compiled with MKL bindings

^e at least we have not been able to

TABLE 4.1: Comparison of how well different libraries and interfaces to use basic dense linear algebra in C++ fit our requirements. The symbols used mean “✓” for “yes / very good”, “✓” for “somewhat / quite good”, “✗” for “somewhat / rather bad” and “✗” for “no / very bad”.

4.1.2.5 Comparison

Table 4.1 compares the four options discussed thus far with respect to the requirements we have stated in the introduction of this section.

To get acquainted with their APIs and to be able to make a sound decision, we have implemented small benchmarks for the four libraries as well as a naïve self-made C++ implementation (lst 4.5). Our minimum requirement for any external library was that it be at least as fast as the naïve implementation.

Figure 4.1 shows how well the libraries perform with regard to equation 4.1. It can be seen clearly from the plot that the libraries can be classified into the two groups “fast” and “slow”. The “slow” group is bounded below by our self-made loop which is what we expect. In addition, the TNT library and uBLAS fall into it. The “fast” group is lead by using the MKL directly via the CBLAS interface which is not very surprising either. The uBLAS and Eigen libraries with MKL bindings are somewhat less efficient than using CBLAS directly. Note how close Eigen’s performance comes to the MKL.

Figure 4.2 shows the same plot for a parallel run using 32 CPUs. Our self-made solution is again hopelessly outperformed which shows that the libraries include heavy optimizations. The abstraction penalty for using the MKL via uBLAS or Eigen seems to be larger in the parallel case and Eigen’s own code is not as competitive as in the sequential case. Please note how large the matrices have to be for all libraries to achieve maximum performance in parallel. This will also be important for our own work.

```

#include <algorithm>
#include <cstddef>

#ifdef restrict
#define restrict /* empty */
#else
#define restrict
#endif

void
example(const std::size_t n, const std::size_t k, const std::size_t m,
        const double *const restrict a_ptr,
        const double *const restrict b_ptr,
        double *restrict c_ptr,
        const double alpha, const double beta) noexcept
{
    if (beta == 0.0)
    {
        // C might be uninitialized and even if not, filling with zeros
        // will be faster than multiplying each element with 0.
        std::uninitialized_fill(c_ptr, c_ptr + n * m, 0.0);
    }
    else
    {
        for (std::size_t idx = 0; idx < n * m; ++idx)
            c_ptr[idx] *= beta;
    }
    for (std::size_t l = 0; l < k; ++l)
        for (std::size_t j = 0; j < m; ++j)
            for (std::size_t i = 0; i < n; ++i)
                c_ptr[i + n * j] += alpha * a_ptr[i + n * l] * b_ptr[j + m * l];
}

```

LISTING 4.5: Our self-made reference implementation of computing the benchmark expression $\alpha AB^T + \beta C$ and store the result in C . Matrix data is assumed to be in column-major order. The loops can be parallelized with OpenMP, however, for the second, one has to be careful not to introduce a data race. The macro `restrict` can be `#defined` to a compiler-builtin with the semantics of the corresponding C keyword. For example, it would be `#defined` to `__restrict` for the GNU C++ compiler. For compilers that do not support such an extension, the “keyword” simply is “`#defined` away”. (The GNU Autotools are actually a little smarter and take care not to `#define restrict` if the compiler already treats it like a keyword.)

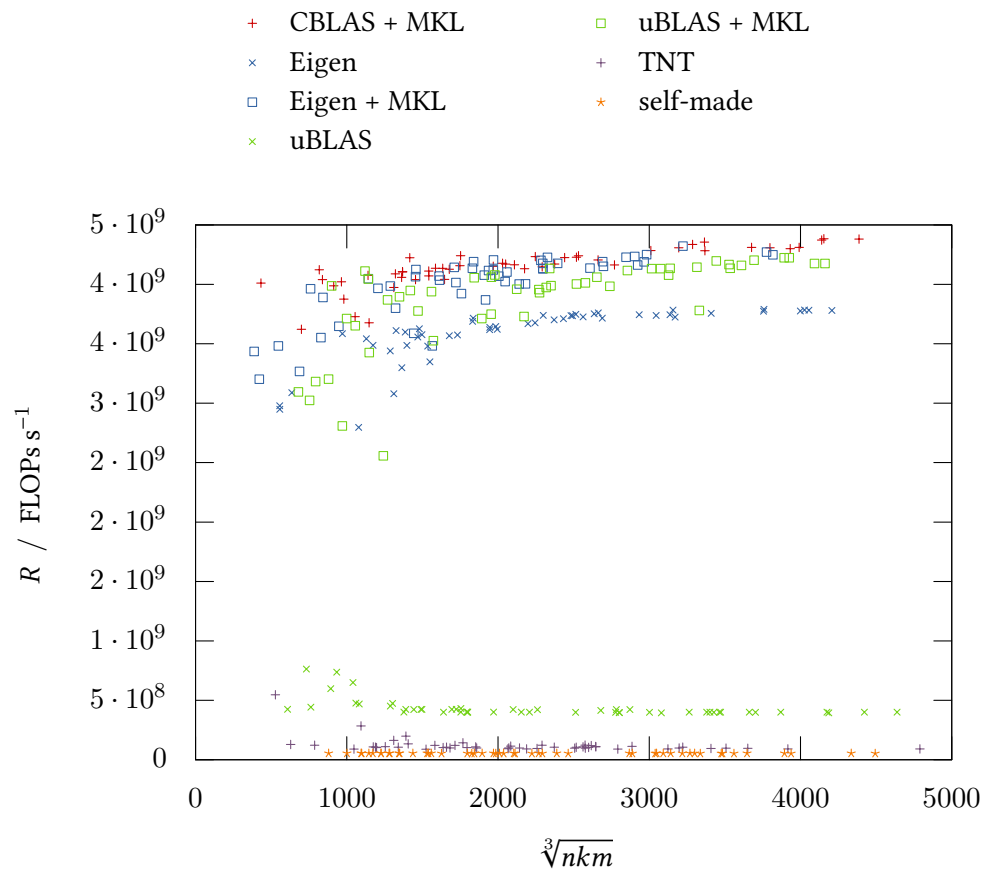


FIGURE 4.1: FLOP rates of the compared linear algebra libraries during sequential execution. Plotted is the FLOP rate against the “hypothetical” matrix size for computing the expression $\alpha AB^T + \beta C$ where A is a $n \times k$, B a $m \times k$ and C a $n \times m$ matrix. We assume that $W_{\text{eff}} = nkm$, corresponding to $\omega = 3$. The machine used for this experiment is identical to the ITI-120 described later in this work (see tab 5.1).

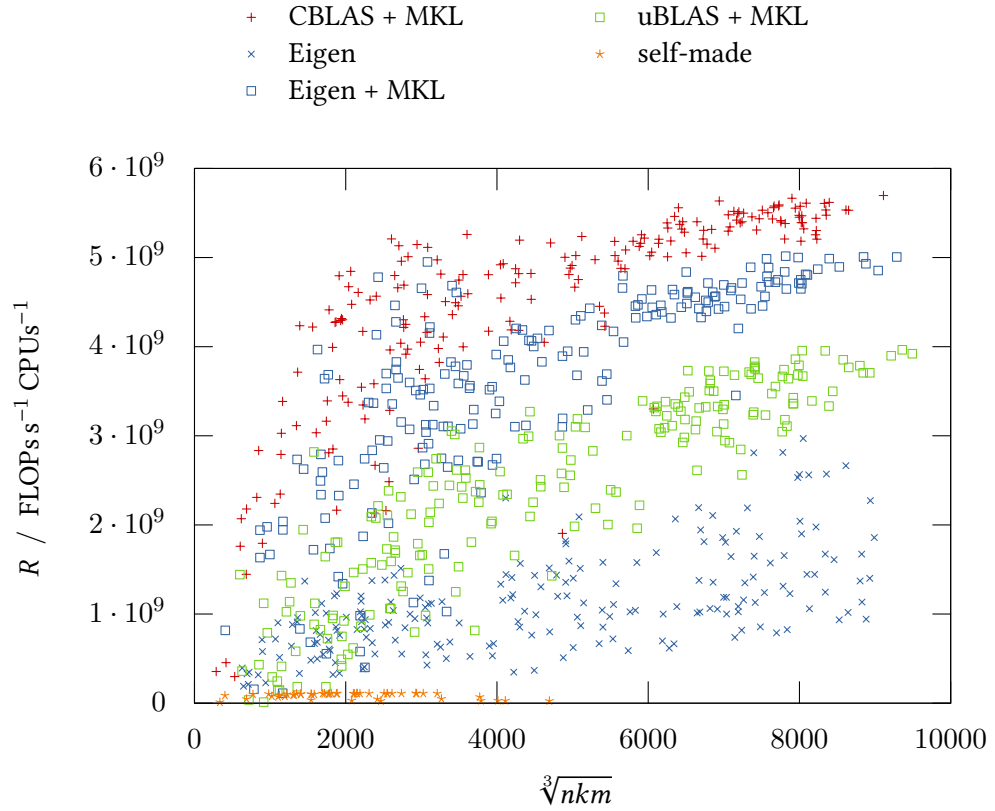


FIGURE 4.2: FLOP rates of the compared linear algebra libraries during parallel execution. Plotted is the FLOP rate against the “hypothetical” matrix size for computing the expression $\alpha \mathbf{AB}^T + \beta \mathbf{C}$ where \mathbf{A} is a $n \times k$, \mathbf{B} a $m \times k$ and \mathbf{C} a $n \times m$ matrix. We assume that $W_{\text{eff}} = nkm$, corresponding to $\omega = 3$. This experiment was conducted on a machine with 32 CPUs and Advanced Vector Instructions. (It is the ITI-127 machine, described later. See table 5.1 for details.)

The two benchmarks have been recorded on different machines so please don’t compare their absolute values.

Putting everything together, we decided to implement our algorithms on top of Eigen as it would be the best fit for our requirements. For benchmarks we run on the machines at our institute where Intel’s proprietary MKL is already installed, we would bind to it via a simple configuration switch to get maximum performance and parallelism. Yet, we (or anyone else) can build our software without any non-free dependencies by default and still get very good performance and parallelism (better than with any other option, that is). The elegance and maturity of Eigen further contributed to that decision.

4.1.3 Tools and Programs

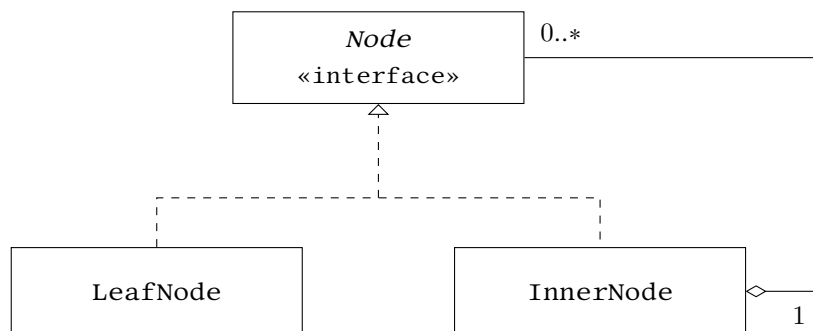
We have developed the code on 64 bit GNU/Linux systems using the GNU tool-chain. We have used the following tools to support our development work. All of them are free software.

- The *GNU Compiler Collection (GCC)* – most notably, its C++ front-end *g++* – was used to compile our code. Of course we have also used the associated preprocessor, assembler, linker and system libraries. We have used versions 4.9.1 and 4.8.2⁸.
- *Gprof* is a profiling tool that accompanies GCC. We have used it to detect hot-spots in our code and assure ourselves that the bulk of work is done inside library calls.
- *Gcov* is a code coverage measuring tool for GCC. We have used it to assess the unit test coverage of our code.
- The *GNU Debugger (GDB)* was used to hunt bugs in our code.
- *Valgrind* is a special-purpose debugger with a module (*Memcheck*) targeted at detecting memory errors. We have used it to make sure our code is free from leaks and obvious memory errors.
- The *GNU Autotools* are a framework to create very portable and highly automated build systems. We have used them for all our building and packaging, starting from compiling the code over creating distribution tarballs up to typesetting this document. The various POSIX tools play an essential role in this system. We are also using a number of handy scripts the author has developed over the years. Those are included in the distribution tarball.
- The *Subversion (SVN)* revision control system was used to manage our source code.

4.2 Algorithms and Data Structures

The most challenging task we have dedicated the bulk of our implementation effort to was designing data structures that efficiently model the structure of hierarchical matrices and formulate algorithms for basic linear algebra operations on them. Of course, these algorithms should internally call into the third-party

⁸The lower of these is likely the oldest version that will work since we have developed the code with the newer one and then added just so many workarounds to make it compile with the older one. Due to our intensive use of modern C++11 features, it is likely that there are many more issues with older compilers, especially those that have been released before the C++11 standard. While we are using new features liberally, we emphasize that our code is strictly standards compliant to our best knowledge and should be compileable with any conforming compiler (of which there is none, unfortunately).

FIGURE 4.3: UML diagram of the *Composite Pattern* [5].

library (Eigen, in our case) that is used for classical dense linear algebra. We have aimed to make these structures as high-level as possible and abstract away as many of the details as possible from the user. Once this was done, the actual algorithms could be implemented in just a few lines of very high-level code that looks almost identical to the pseudo code listings presented in chapter 3. C++ aids this task very well with polymorphism, operator overloading and argument dependent lookup (ADL) [22, § 14.2.4].

4.2.1 Data Structures for Hierarchical Matrices

The hierarchical structure of an H-matrix calls for an implementation using the *composite* pattern [5]. In that pattern, a type recursively refers to itself via an abstract interface. The pattern is the natural implementation of trees in object oriented languages and visualized in figure 4.3.

4.2.1.1 The Abstract `HMatrix` Class

For our implementation, we defined an abstract `HMatrix` class. It supports a very rudimentary interface. Apart from knowing its own size, it provides operations to access individual elements. These must of course only be used for debugging purpose since having multiply delegated virtual function calls for each element access is clearly unacceptable in high-performance applications. On the other hand, these functions allowed us to get started quickly with our high-level algorithms (that then were of course terribly slow). Once we saw that they were working, we successively optimized our low-level algorithms until no inefficient operations were performed any more. The other two operations specified by the `HMatrix` interface are a `clone` operation that returns a deep polymorphic copy (since polymorphism prevents us from using copy constructors here) and a `passOverRegion` function that implements a visitor-like interface but without

double dynamic dispatch⁹.

4.2.1.1.1 Visitors The visitor mechanism is the foundation for practically all algorithms we define on `HMatrixes`. Its semantics are straight-forward: Often, an operation on or with a matrix can be formulated in terms of all its blocks of non-zero elements. To do this efficiently, the matrix class had better tell us where its non-zero blocks are so we waste no time dealing with the zero blocks.

The `ActiveVisitor` interface specifies exactly one function: `visitRegion`. It takes as arguments a reference to an `HMatrix`, a flag that indicates whether this matrix is to be interpreted as its transpose, four coordinates defining a rectangle inside the matrix and two more coordinates defining the offset of that rectangle inside whatever is the whole matrix.

To apply an operation to all regions of an `HMatrix`, one creates an `ActiveVisitor` object that overrides the `visitRegion` function accordingly and then passes the visitor to the `HMatrix`' `passOverRegion` function. This function takes additional parameters to specify whether the whole matrix or only a slice of it should be shown to the visitor. In addition, it has parameters that allow propagation of the information whether the matrix is to be interpreted as its transpose and what the coordinates of the region's origin are in whatever is the global coordinate system. The matrix is now responsible for calling the visitor's `visitRegion` function exactly once for each of its non-zero blocks. The order is unspecified. Of course, if the matrix is composite, then it will pass the visitor on to its children until recursion ends at the leaf matrices where the visitor is finally applied.

Sometimes, it is not necessary to modify the visited matrix. In this case, it is undesirable from a software engineering point of view to have to use a mechanism that is formulated non-constant. Therefore, there also is the `PassiveVisitor` interface. It is identical to the `ActiveVisitor` interface except that the `visitRegion` function receives a `const` reference to the visited regions.

A simple use-case for an `ActiveVisitor` (that does not leverage the full power of this mechanism) is an operation to scale an `HMatrix`' elements by a scalar factor. The visitor would store the factor as a member and then, for each non-zero region it is shown, call a library function to actually scale the elements.

An example where a `PassiveVisitor` could be useful is if an `HMatrix` should be "flattened". That is, all elements should be copied into a single huge two-dimensional array. Since we do not need to modify the original matrix, a `PassiveVisitor` is sufficient. For each region it is shown, the visitor performs some rather simple coordinate transformation to figure out where it is located in the "flat" matrix and then copies over the elements (preferably via a library call) into the array it has stored as a pointer member.

⁹This might be considered a design flaw and if we were to implement the thing again, we would do the *Visitor* pattern correctly. Fortunately, since we have only one leaf node type, the tyranny of dominant decomposition is rather mild in our case. However our code *is* littered with `if (auto full_ptr = dynamic_cast<FullMatrix<T>*>(&matrix)) { ... }` constructs, many of which could have been avoided if double dynamic dispatch had been used.

Both `ActiveVisitors` and `PassiveVisitor` are passed around by-reference so they can carry state. For a `PassiveVisitor`, this is often essential. For example, if we want to find the maximum element in a matrix.

4.2.1.2 Leafs: The `FullMatrix` Class

The only leaf node type we have derived (except for a `MockHMatrix` class for unit testing) is the `FullMatrix` class. It is a rather simplistic wrapper around the `Eigen::Matrix` class to which it directly delegates calls to the size and element-wise accessor functions.

The `FullMatrix` class extends the `HMatrix` interface by a `getRawMatrix` function that returns a reference to the underlying `Eigen::Matrix`. A `FullMatrix` does not know anything about its structure. Visitors are therefore passed over the entire matrix.

In order to `clone()` a `FullMatrix`, a copy of the underlying raw matrix is wrapped inside a new `FullMatrix`.

4.2.1.3 Nested Dissection: The `BlockMatrix` Class

The `BlockMatrix` class¹⁰ represents the special structure of a matrix obtained via nested dissection (§ 2.4.1). It has references to seven child `HMatrix`s with some restrictions on their relative dimensions dictated by the structure of the matrix. We did not make our implementation symmetry-aware which means that we store all blocks separately, even if they contain the same data¹¹. The size of a `BlockMatrix` (which must be square) is defined by the sum of the sizes of its diagonal blocks. Element-wise access is readily delegated to the respective child matrix by bracketing the indices along both dimensions. Visitors are passed over the seven children leaving out the two off-diagonal zero blocks. A clone of a `BlockMatrix` is made by recursively `clone()`ing its children and arranging them into a new `BlockMatrix`.

`BlockMatrix`s extend the `HMatrix` interface by providing access to the seven sub-matrices via functions `getBlockA`, ..., `getBlockY`. (The names are the same as introduced in section 2.4.1.) These functions return a pointer to the respective sub-matrix or a `nullptr` if the `BlockMatrix` is a degenerate 0×0 matrix with no children.

4.2.1.4 Arbitrary Children: The `ArrangedMatrix` Class

The `ArrangedMatrix` class is the most general case of an `HMatrix` node. It has an arbitrary number of arbitrarily arranged children. The only restrictions

¹⁰In retrospective, `NMatrix` would probably have been a better name.

¹¹Avoiding this turned out to be harder than expected. While reading access is easily mapped to the transposed region, it is not so clear what should happen if an entry is written to. For example, is the algorithm to scale a matrix with a scalar factor supposed to know it should visit only half the matrix? We have postponed this decision until no time soon.

are that the child matrices must fit the boundaries of the `ArrangedMatrix` and no two child matrices must overlap with each other. The matrix is represented as an unordered list of `MatrixAnchors` where a `MatrixAnchor` is a very simple data structure storing a pointer to an `HMatrix` (the child matrix) and two indices specifying the coordinates of the child matrix' first element inside the whole `ArrangedMatrix`.

To access a single element inside an `ArrangedMatrix`, a linear search over the list of children is required. For each child, it has to be checked whether the requested index falls into their region. If no child is tested positively, then the index falls into a zero block.

Likewise, if a visitor is to be passed over a region of an `ArrangedMatrix`, we check for each child whether it overlaps with the requested region and if so, recurse into the respective sub-region of the child. This might sound worse than it is because the most frequent case is that the entire matrix is to be passed over in which case we need to deal with each child individually, anyway. In addition, the number of children is usually not very large.

An `ArrangedMatrix` is `clone()`d by `clone()`ing all its children and inserting them (at the same positions) into a new list of `MatrixAnchors` in a new `ArrangedMatrix` with the same dimensions.

`ArrangedMatrixes` provide access to their children via an iterator over the `MatrixAnchors`.

`ArrangedMatrixes` are general and flexible but sometimes, more is known about the structure and this extra information can be exploited by certain algorithms. Therefore an `ArrangedMatrix` keeps three flags to indicate whether it is *row-linear*, *column-linear* and *diagonal*. Usually, a matrix can only have one of the properties except in trivial cases (such as an empty matrix). Table 4.2 explains these three properties further.

4.2.1.5 Slicing and Transposing: The `MatrixView` Class

The `HMatrix` classes discussed so far all own their data and are therefore heavy-weight components. Often times, we want to refer to a slice of another matrix as a matrix in its own right or view a matrix as its transpose. This leads us to the introduction of a light-weight `MatrixView` class. A `MatrixView` object is only a few words in size storing a pointer to the viewed `HMatrix`, four indices defining the region that is viewed and a flag whether the view is transposed.

In order to be able to create a view of `const` and `non-const` `HMatrixes` alike, we need two types of views: `ConstMatrixViews` and `MutableMatrixViews`. These are actually just template aliases for the general `MatrixView` class that takes an additional boolean template parameter that selects whether the viewed matrix may be modified.

Element-wise access on a `MatrixView` involves adding an offset to both indices and – if the view is transposed – eventually swapping row and column index. The thus obtained new indices are passed on to the viewed matrix. Visitors are ap-

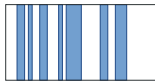

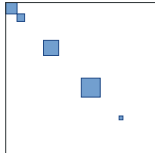
Property	Definition	Example
row-linear	Each sub-matrix has the same height as the entire matrix.	
column-linear	Each sub-matrix has the same width as the entire matrix.	
diagonal	Each sub-matrix is anchored at the main diagonal.	

TABLE 4.2: Special properties of `ArrangedMatrixes`. It is readily seen that a matrix is row-linear if and only if its transpose is column linear. Since an empty `ArrangedMatrix` (with no children) fullfills all three properties in a trivial sense, each `ArrangedMatrix` starts with all three flags initially set. Each time a tile is added, it is checked (with constant overhead) for each property that is still set whether this new tile voids that property. Querying whether an `ArrangedMatrix` has any of the properties is a constant operation that simply returns the value of the respective flag.

plied to the viewed region of the underlying `HMatrix` only with the flag whether the region is transposed xor'ed with the transpose flag of the view.

Unfortunately, it is not possible to implement `clone()` properly for `MatrixViews`. If the clone were shallow, it would violate the requirement for the operation to have copy-semantics. If it were deep as required by the interface, it could not even return the same type as the cloned object. Therefore, `MatrixView::clone` is amputated (overridden to unconditionally throw an exception). Likewise, for obvious reasons, the `ConstMatrixView::set` member had to be amputated, too.

4.2.1.6 HVector and FullVector

For vectors, we also have an abstract `HVector` interface that merely defines functions to query the size of the vector and element-wise access. However, the only implementing class is `FullVector` that delegates to `Eigen::Matrix` again and provides a `getRawVector` member function that returns a reference to it. There is no visitor interface for `HVectors`.

Figure 4.4 summarizes the data types introduced in this section. Equipped with these types, we can stack together any configuration we need. In the following section, we will describe how basic linear algebra operations can be carried out with these types.

4.2.2 Basic Algorithms for Hierarchical Matrices

In this section, we will discuss how some basic operations can be implemented for the data structures introduced in the previous section. We have already explained how trivial element-wise operations such as scaling with a scalar factor can be implemented using `ActiveVisitors`. In this section, we will focus on two operations that are not so obvious to implement and have received the most attention on our behalves: sums / differences and products of `HMatrixes`.

4.2.2.1 Implementation Strategy

Implementing a fully-fledged library of linear algebra operations would have been both, overkill for our task and impossible within the time constraints for this work. Instead, we have focused on making those operations efficient that are actually used by our program and left the others alone.

Since we were not sure from the beginning how the complete program can work, we have adopted a “tracer bullet” approach [9]. First, we've implemented naïve algorithms for all operations and wrote black-box unit tests for them. This allowed us to quickly formulate our higher-level algorithms using arbitrary expressions and see whether they will work correctly. At the beginning of every such algorithm we put a macro `BSC_KLAMMLER_INEFFICIENT_ALGORITHM` that serves two tasks. First, it is a prominent warning that the algorithm is not optimized yet. Second, it allows us to inject tracer code to detect what algorithms are called by our program. The `BSC_KLAMMLER_INEFFICIENT_ALGORITHM` macro

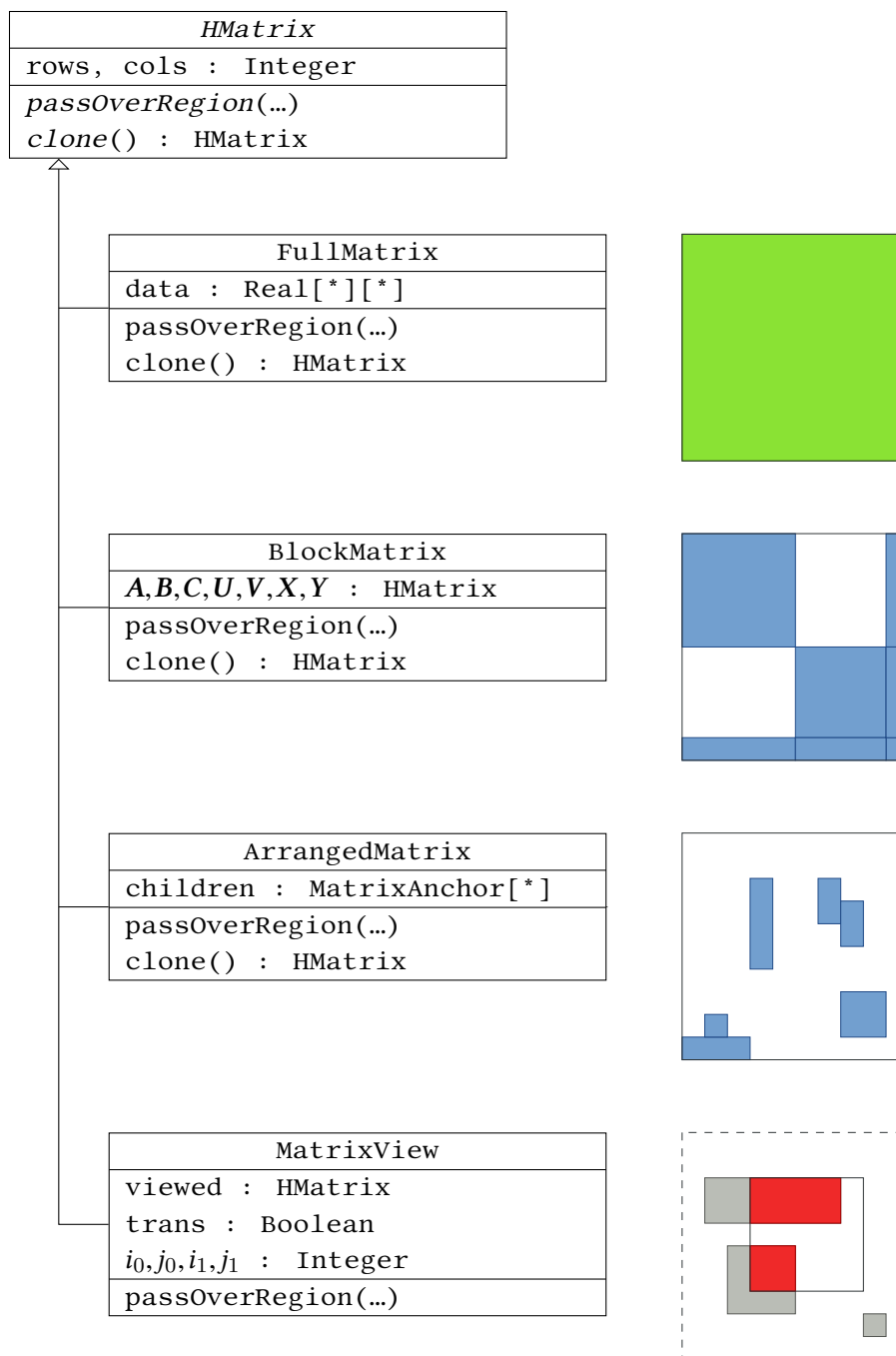


FIGURE 4.4: UML class diagram of the various matrix classes.

is defined to invoke a function that – depending on a global policy variable – may act in different ways.

- If the policy is `IGNORE` then the function call is a no-op. This is useful during early states of development where we simply want to get going.
- A policy of `WARN` will log a warning message that the inefficient algorithm was called. This helps deciding what algorithms still need optimization once the basic scaffolding stands.
- The `THROW` policy triggers a fatal error. This is used when benchmarking the code to make sure all calls to inefficient functions have been optimized away.

Once we were confident that our general design would work, we started turning on the warnings and optimized one operation at a time. While doing so, we could run regression tests against the unit test base as well as verify that our high-level algorithms still work correctly with the optimized routines as often as we wanted. Since optimizing often meant writing additional functions, we also needed additional white-box unit tests that exercise them. We have used `Gcov` to verify on a regular basis that at least, we reach almost complete instruction coverage.

For some operations (such as matrix-vector products) we have been able to optimize the general version of the operation. For other operations, however, this was not simple and we have ended up optimizing only those special cases that are hit by our program and resort to the naïve implementation if none of the special cases matches.

All these operations are implemented as non-member non-friend functions in a single header file that is kept separately from the `HMatrix` class definitions.

4.2.2.2 Sums and Differences

Obviously, the logic for adding and subtracting `HMatrixes` is the same. Therefore, all algorithms that deal with them are implemented as templates that take an additional parameter that specifies the operation. The user-level operator overloads might then (parameter validation aside) look like this.

```
template<typename T>
HMatrix<T>&
operator+=(HMatrix<T>& self, const HMatrix<T>& other)
{
    algorithms_hpp::plus_minus_to<'+>(self, other);
    return self;
}
```

where `algorithms_hpp::plus_minus_to` is an internal function that does the actual work. Therefore, the code that needs to be duplicated for operator `-=` is negligible. `plus_minus_to` is defined as

```

template<char Op, typename T>
void
plus_minus_to(HMatrix<T>& self, const HMatrix<T>& other)
{
    static_assert(Op == '+' || Op == '-',
                  "template parameter 'Op' must be '+' or '-");
    // ...
}

```

and at some point (where the actual addition or subtraction is carried out) might contain something like

```

switch (Op)
{
    case '+': val += diff; break;
    case '-': val -= diff; break;
    default: BSC_KLAMMLER_NOT_REACHED;
}

```

where the `switch` statement generates no machine code since `Op` is a compile-time constant. (The above snippet is actually copied from the naïve fallback implementation and occurs inside a tight loop there.) The macro `BSC_KLAMMLER_NOT_REACHED` is defined to unconditionally trigger a fatal error and can be placed as an assertion at points the control flow should never reach.

This implementation is a little more complicated than simply defining operator `-=` in terms of

```

template<typename T>
HMatrix<T>&
operator-= (HMatrix<T>& self, const HMatrix<T>& other)
{
    return self += -other;
}

```

but while it is possible to optimize the negation away here, reliably doing so would require more thoughts than a few `switch(Op) { ... }` here and

The application of the update is performed in two logical steps. For each non-zero block in the addend (or subtrahend), it is checked where it belongs in the augend (or minuend). There are at least three cases to distinguish.

- *The update is completely confined inside a non-zero block of the augend (or minuend).* In this case, a simple library call to apply the update is sufficient.
- *The update falls completely inside a zero block of the augend (or minuend).* It depends on the type of the `HMatrix` whether we can handle this case at all. For an `ArrangedMatrix`, we simply add a new tile with just the data of the update (or its negative in case of subtraction). For a `BlockMatrix` on the other hand, this case is a contract violation as such a matrix must never contain anything inside the zero blocks that are defined by its structure. If

we get such an update for a `BlockMatrix`, we have no chance but must raise an error.

- *The update partially overlaps with a non-zero block in the augend (or minuend).* This case is the most difficult one. If the augend (or minuend) is not an `ArrangedMatrix`, the operation is invalid in the first place anyway. However, even if we *are* updating an `ArrangedMatrix`, it is not clear how we should proceed. Adding tiles for the non-overlapped regions would quickly cause fragmentation of the matrix. A clever heuristic to decide how to grow / merge tiles would be complicated and cause additional overhead. Since this situation never occurs in our algorithms (by design, cf chapter 3) we have simply disallowed it.¹²

The above logic is implemented via a pair of visitors. A `PassiveVisitor` visits the addend (or subtrahend) to extract all non-zero blocks. Those it passes to an `ActiveVisitor` that passes over the augend (or minuend) to find out which of the above three cases applies and handle it properly.

To add and subtract `HVectors` – since there are only `FullVectors` – we can always call directly into the library. Only if one of the vectors is an unknown type (which only happens if a mockup is passed for the unit tests) a naïve copy loop takes over.

4.2.2.3 Products

4.2.2.3.1 Matrix-Vector Products Since it is much simpler, let us discuss the product of an `HMatrix` and a `FullVector` first. The result will also be a `FullVector`.¹³ Assume for the sake of this discussion that the matrix is the left-hand and the vector the right-hand operand. The other case is easily implemented in terms of this one simply by swapping the operands. If one were to distinguish between row and column vectors – which we don't – an additional transposition would have to be done.

We observe that no matter how the matrix looks, the result will be the sum of the vector multiplied individually with each non-zero block in the matrix. Those individual products yield vectors where all elements above and below the non-zero block are zero. These products are computed straight-forwardly by passing a `PassiveVisitor` over the `HMatrix` that multiplies each non-zero block with the respective piece of the vector. The result is accumulated in a `FullVector` that is initially filled with all-zeros. This procedure is illustrated in figure 4.5.

¹²If it would happen, the naïve fallback algorithm will kick in and trigger an error at the first attempt to add (or subtract) a non-zero element inside a zero block. The additional check for each element whether it is zero is needed in the naïve version so it can treat everything as a `FullMatrix`.

¹³The case that the vector is not a `FullVector` again only applies to unit test mockups. The result will still be a `FullVector` in this case.

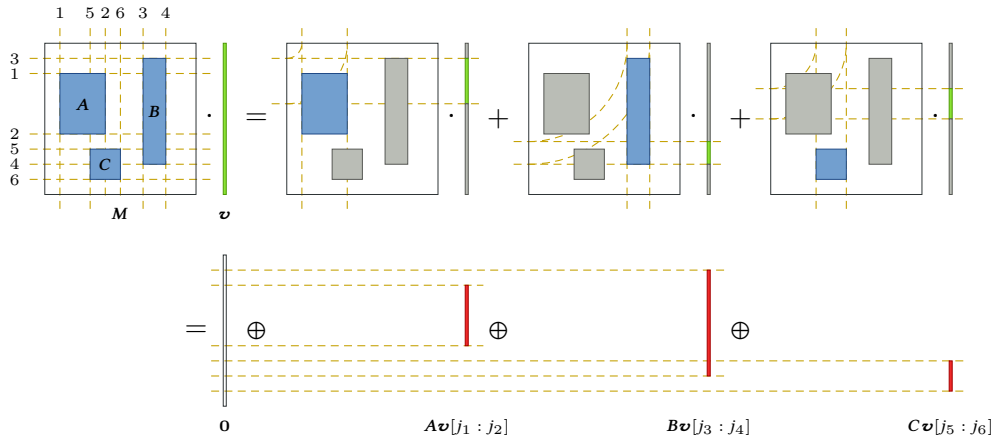


FIGURE 4.5: Product of an HMatrix M and an HVector v that is treated as a FullVector. For each non-zero block of M , we compute the product of that block and the respective slice of v that spans the same indices than the block’s columns. Then we add together the partial results where we have invented the operator “ \oplus ” to mean “add at the right offset”. The small numbers above and left to the matrix M number the row and column indices that define the slices of the non-zero blocks. That is, $A = M[i_1 : i_2][j_1 : j_2]$, $B = M[i_3 : i_4][j_3 : j_4]$ and $C = M[i_5 : i_6][j_5 : j_6]$.

4.2.2.3.2 Matrix-Matrix Products Unfortunately, the above approach grows unwieldy if the right-hand operand is a matrix too. Even if it were a FullMatrix – which more often than not, it is not – it would be inefficient to implement a matrix-matrix product as a bunch of matrix-vector products. We have only implemented the following special cases.

- *Both operands are a single FullMatrix.* This case is a simple call into the dense linear algebra library. The result is a FullMatrix again.
- *The left-hand operand is a column-linear ArrangedMatrix.* The result is again a column-linear ArrangedMatrix with sub-matrices of exactly the same size and at exactly the same positions than in the left-hand operand.
- *The right-hand operand is a row-linear ArrangedMatrix.* The result is again a row-linear ArrangedMatrix with sub-matrices of exactly the same size and at exactly the same positions than in the right-hand operand.
- *The left-hand operand is a column-linear ArrangedMatrix and the right-hand operand is a row-linear ArrangedMatrix.* The result is an ArrangedMatrix with a “grid” of sub-matrices of sizes and positions defined by the sizes and positions of the sub-matrices in the operands.
- *Both operands are arbitrary HMatrixes.* The result is computed by an inefficient naïve element-wise loop and returned in a FullMatrix, throwing away any structural information. This case never happens in our code.

While this case is simple to compute, it requires a little more work to detect. This is because the `FullMatrix` might be buried under layers of other `HMatrixes` so simply checking the type of the operands (either statically or dynamically) is not sufficient. The most obvious example how this can happen is if a `MatrixView` is created. While the matrix data is still a single dense block, the dynamic – leave alone with the static – type of the operand is not `FullMatrix`. An other example would be a degenerated `ArrangedMatrix` that has exactly one `FullMatrix` of equal size as child. Of course, this situation can be made arbitrarily complex by stacking all kind of `HMatrixes` together as long as the final view projects to a sub-matrix of a single `FullMatrix`.

Our approach to this problem is to pass a `PassiveVisitor` over the operand. The first non-zero block that is shown to this visitor is recorded (a pointer to it, four indices that define the sub-block and a flag whether the block is transposed). If a second block is shown to the visitor afterwards, it is invalidated. If after passing over the operand, the visitor has seen exactly one block and that block has the same size as the operand, we use it for the multiplication. otherwise this algorithm bails out and the naïve substitute acts as a fallback. If a single dense block of full size could be extracted from both operands, the Eigen library is called to compute their product.

The other three cases are really just creating an empty `ArrangedMatrix` of appropriate dimensions for the result, and then looping over all combinations of sub-matrices, computing their respective product and adding the result as a sub-matrix in the overall product. Figure 4.6 illustrates this.

Often times, we are dealing with a matrix that is row- or column-linear but doesn't say so. This happens for example in our algorithms when we slice out the M_{21} part (see § 3.2.1) of a `BlockMatrix`. While the slice is of type `MatrixView` that has no concept of linearity, we as the user know that the matrix actually is column-linear. Therefore, we use a trick to get the object learn its structure. Instead of slicing out a `MatrixView`, we are creating a new `ArrangedMatrix` where each sub-matrix is a `MatrixView` of a single dense non-zero block.

To do so, we define the static factory function `makeArrangement` that takes as parameters an `HMatrix` to slice, four coordinates that define the region to slice out and a functor to extract a non-zero block. This function creates an empty `ArrangedMatrix` of the same size as the region to be sliced out and then passes a `PassiveVisitor` over that region that, for every non-zero block it is shown, passes the block to the functor and inserts the returned `HMatrix` into the new `ArrangedMatrix` at the correct position. Finally, the new `ArrangedMatrix` is returned. In effect, this “flattens out” a general `HMatrix` to yield an identical `HMatrix` with recursion depth one.

This mechanism is very flexible and could be applied to many problems. For the issue described above, we use a functor that does not `clone()` the non-zero blocks – which would certainly be a terrible idea – but creates a `ConstMatrixView` of them. The storage needed for a single `MatrixView` and its entry inside the `ArrangedMatrix` is about a dozen machine words so the overall cost for the

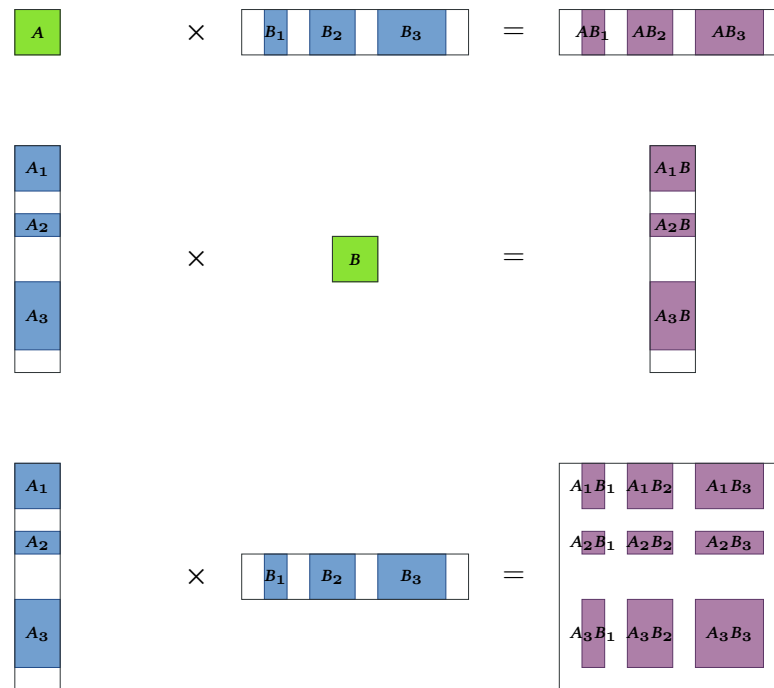


FIGURE 4.6: Products of row- and column-linear ArrangedMatrixes.

operation can be reasonably neglected. Once the arrangement is made, it knows whether it is column-linear so the fast multiplication technique will work. (Note that every sub-matrix of the arrangement is a `ConstMatrixView` to a `FullMatrix` so the individual matrix-matrix products will be able to use the simple and efficient dense case.) The overhead for creating the arrangement first is additionally compensated by the fact that traversing the non-zero blocks of the `ArrangedMatrix` really is an iteration with a constant recursion depth of two (One for the `ArrangedMatrix` itself and one for the contained `MatrixViews`.) so it will be more efficient than traversing a deeply recursive `BlockMatrix`. This comes in handy as we can use the arrangement for more than one product in our algorithms.

4.2.3 Decompositions

Equipped with the data structures and basic algorithmic building blocks described in the previous section, the block LL^T and LDL^T decomposition algorithms could be implemented almost as the pseudo code listings. Since the logic for the parallel recursive processing and concurrency-safe update handling and propagation is shared between both algorithms, we have derived them from a common base `RecursiveSPDPreconditioner`. This class is in turn derived from the even

more general `Preconditioner` class.¹⁴ These form a template (in the sense of Gamma et al. [5]) class where we can plug in Cholesky decomposition or matrix inversion. The most important aspects of this hierarchy are illustrated in figure 4.7.

The `initImpl` function member of the `RecursiveSPDPreconditioner` basically implements algorithm 3.1 while the actual factoring and update computation logic is delegated to the purely virtual `decomposeImpl` member function. The `BlockLLTPreconditioner` and `BlockLDLTPreconditioner` classes override this template function implementing algorithm 3.2 and 3.3 respectively.

4.3 Parallelization

We distinguish three levels of parallelism (each including all of the previous).

- 0 *none* – The algorithms perform purely sequential.
- 1 *structural* – The recursive descent is parallel.
- 2 *expression-level* – Independent operations in a single expression are computed in parallel. For example, the individual sub-products in figure 4.6 could all be computed in parallel. Unfortunately, we have not implemented this yet due to time constraints.
- 3 *data-level* – The external linear algebra library operates in parallel on the same data.

The parallelization level can be selected at compile-time by defining the macro `BSC_KLAMMLER_PARALLEL` to the respective integer. Of course, if Eigen is to be bound to the MKL, then for level-3 parallelism, the parallel version of it must be linked to and for all other levels the sequential one.

To recurse in parallel, a new thread is created for each dependency sub-tree and joined again before returning to the parent level. Calling the parallel versions of library functions does not require any action on our behalf. However, we did not want to start more threads than we have hardware processing units. Otherwise, the performance might be compromised badly by repetitive context switches for no good reason. Therefore, we have made the number of threads to fork a run-time configuration feature. For the external libraries, we can use the OpenMP `omp_set_num_threads` function to set this number accordingly.

A second consideration is that if structural parallelism has already forked a thread for each processor to be busy, there is no point in calling parallel library

¹⁴We have chosen the name “preconditioner” since originally we have assumed that computing the inverse will introduce such large numeric errors that the result cannot be used immediately but might be useful for preconditioning iterative solvers like the conjugate gradient method. After it turned out that the errors are reasonably small, we have abandoned that idea but still kept the name.

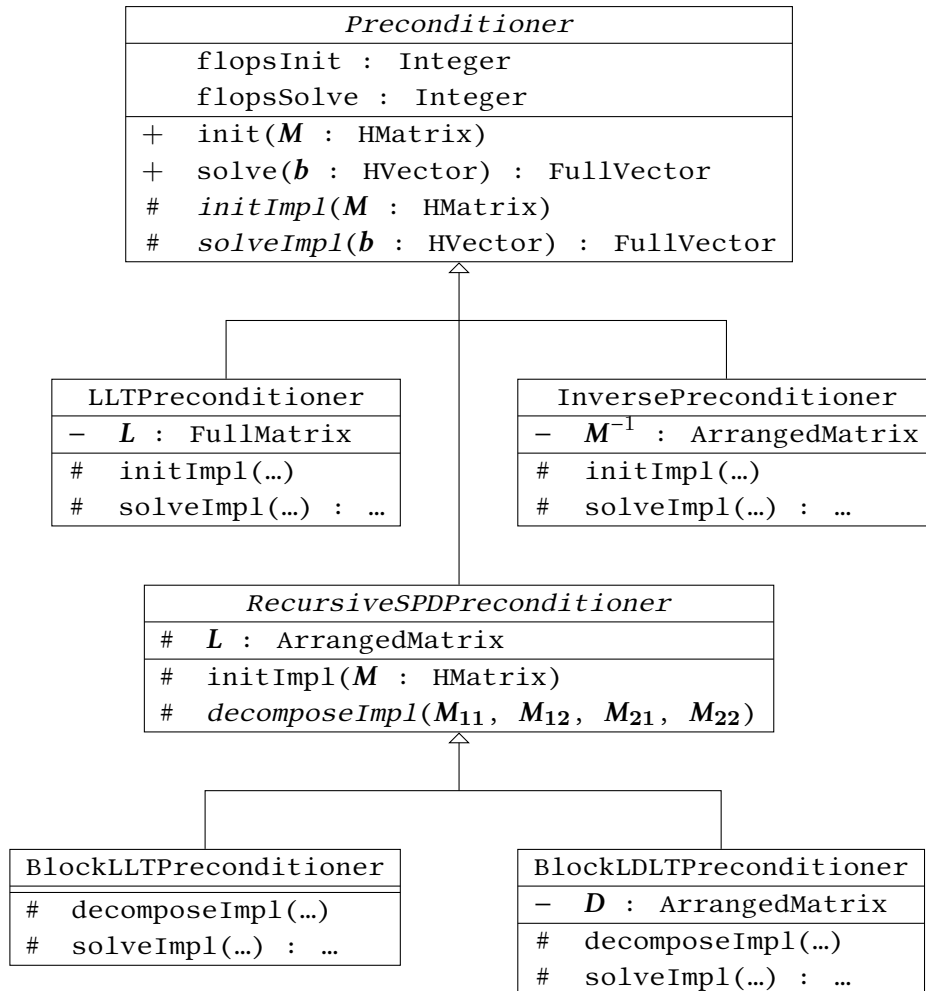


FIGURE 4.7: UML class diagram of the Preconditioners used in our code. The LLTPreconditioner and the InversePreconditioner compute a dense Cholesky factorization and the inverse of the entire matrix (that is flattened for this purpose) respectively and do not exploit the structure of the matrix in any way. They were used to convince us that our effort was worthwhile. The FLOP counting is discussed in section 4.4.

algorithms. On the contrary, doing so would cause the same undesirable context switches but this time even worse since the heavy instruction pipelining and vectorization machinery found on modern hardware that is used for the low-level numerics needs quite long to reach its peak performance. Also, instead of having two processors operate on the same data in parallel, it is usually much better to have each of them operate sequentially on independent data for data locality and cache consistency.

Since our time frame for implementing this work was limited, we could not implement a perfect parallelization scheme but had to make a compromise. We have therefore decided to implement the following simple approach.

As long as there are more processing units available, recursion forks new threads. Once the number of threads has reached the number of processors, further recursion does not fork any more. (If the number of processors is not a power of two, then an approximation is used.) We call this the “single CPU level”.

Below the single CPU level, library calls operate sequentially on their data. Once recursion has crawled up again to a level where some processing units begin to run out of work, all but one threads are blocked and that single active thread calls parallel library functions to operate on all processors simultaneously. This scheme is illustrated in figure 4.8.

This logic is implemented in the `RecursiveSPDPreconditioner` class and therefore shared between the block LL^T and block LDL^T variant. The parallel recursive descent is implemented using C++11 `std::threads`. This logic is encapsulated by packing each recursive call into a lambda expression that is then passed to a dispatch routine. Below the single CPU level, that routine simply calls the two lambdas. The return value (the unsafe update) is passed around by moving it in and out of the lambda. Initially, we call `omp_set_num_threads(1)` to disable data-level parallelism. As recursion ascends above the single CPU level again, the number of threads is set to the user-selected concurrency and processing continues fully parallel, a single node at a time. To achieve this, the nodes above the single CPU level acquire a `std::unique_lock<std::mutex>` before the start processing and release it upon completion of their work when they notify the next node via a `std::condition_variable`.

4.3.1 Synchronization

The algorithms are designed such that they circumvent write dependencies in the first place avoiding the need for locking (cf § 3.2.4.1). However, since we obviously did not implement L (and D) as a single huge 2D-array but an `ArrangedMatrix`, even adding a new sub-matrix needs protection. This is because internally, the `ArrangedMatrix` has to keep a `std::vector` of its sub-matrices and appending to a `std::vector` is not concurrency safe. Fortunately, the critical section can be minimized to a very small constant number of machine instructions.¹⁵ This

¹⁵Strictly speaking, it is actually worse than that. If the vector needs to be re-sized, a memory allocation – which is a potentially blocking system call – and a copy-loop (logarithmic in the overall

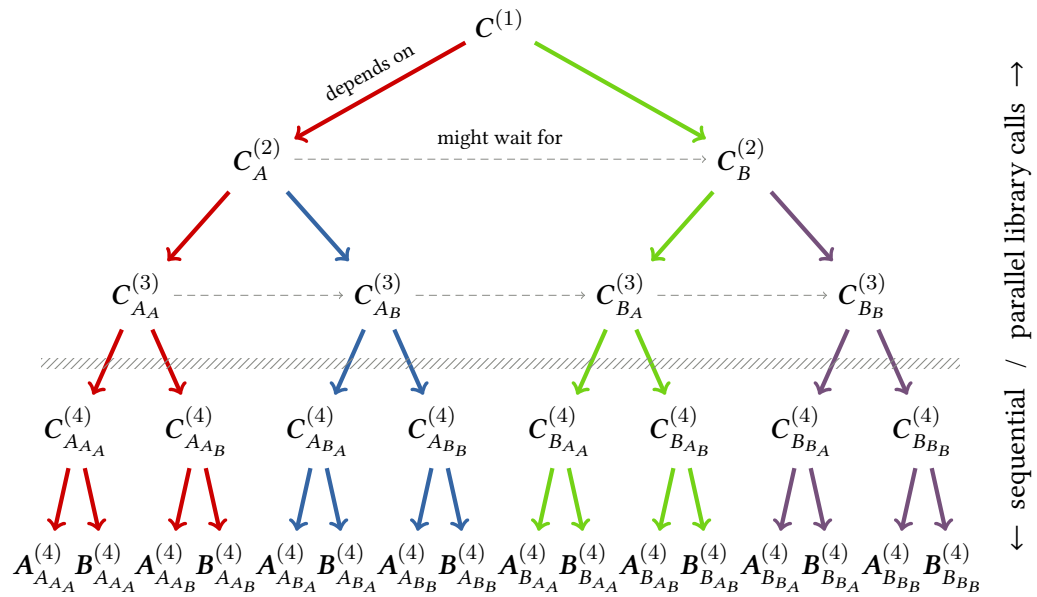


FIGURE 4.8: Parallelization scheme used for the recursive algorithm. This example shows the processing of a matrix with a recursion depth of 4 on a machine with 4 processors. Above the “single CPU level” (hatched bar) each recursion forks a new thread of execution (different threads are marked by different colors), below it, recursion is sequential. Likewise, library calls below the single CPU level are sequential while above it, parallel algorithms are called. On the other hand, the nodes below the single CPU level (if they are on different threads) execute independently of each other, each on its own CPU. In contrast, the nodes above it each occupy all CPUs simultaneously. Therefore, they wait for each other upon ascent so only one node above the single CPU level is active at any time. This waiting is implemented via an ordinary mutex so the order need not be the one indicated by the thin dashed arrows. The symbols indicate the sub-matrix that is factored at the node in question using the syntax introduced in chapter 3. Also compare this figure with figure 3.2.

is done by constructing the new sub-matrix out-of-place and then moving it into the `ArrangedMatrix`. For example, this is how the `BlockLLTPreconditioner` inserts L_{11} and L_{21} into L .

```
full_matrix l11 { /* compute via Cholesky factorization... */ };
arranged_matrix l21 { /* compute via forward substitution... */ };
{
    const auto lck = this->lockL();
    this->getFactorL().addTile(
        std::unique_ptr<h_matrix> {new full_matrix {std::move(l11)}}},
        offset, offset);
    this->getFactorL().addTile(
        std::unique_ptr<h_matrix> {new arranged_matrix {std::move(l21)}}},
        offset + m11.rows(), offset);
}
// Make sure the matrices were actually moved, not copied.
assert(l11.size() == 0);
assert(l21.size() == 0);
```

The `std::unique_ptr<h_matrix>` could be created outside the critical section as well; then only a single move would be required but the code is cleaner this way.

4.4 Observation

The theoretical number of FLOPs needed by the algorithms cannot be expressed in a simple formula since we have left too many degrees of freedom for the inner dimensions of the matrix. Therefore, we have augmented our algorithms to compute them on-the-fly. For this purpose, the `Preconditioner` class has two counters for the flops needed for the decomposition and each solving respectively. These counters are incremented by the algorithms as they see the work. (If more than one system is solved, only the first run will update the counter.) Since the algorithms operate concurrently, the counter must be updated atomically which is done via a lock-free `std::atomic<std::size_t>`. Since the result is only meaningful after the complete decomposition is done anyway, we are using the relaxed memory order for updating the counters so to minimize the overhead. Finally, the preprocessor macro `BSC_KLAMMLER_COUNT_FLOPS` can be `#defined` to zero to disable FLOP counting all together (we did not see any difference when we did this). The number of FLOPs the algorithms report are the theoretical numbers. Our implementations perform more than those since we cannot exploit symmetry up to the theoretical level due to missing library functions.

matrix size) is needed inside the critical section. We'll ignore this here.

4.5 Quality

We have added many bells and whistles to our code to be informed about bugs as much as possible. Therefore, all functions validate their parameters and check their post conditions.

To validate parameters, we check for and report contract violations via throwing `std::invalid_argument` exceptions. The logic required for the checks is wrapped inside conditional statements where we check the value of the preprocessor macro `BSC_KLAMMLER_CHECK_ARGUMENTS`. This helps the human reader telling the business logic apart from the error checking code but also allows us to compile-out the parameter validation all together in builds we use for benchmarking.

For internal checks (pre- and post conditions) we use the standard library's `assert` macro. It can also be made to produce no machine code by defining the preprocessor symbol `NDEBUG`. The same mechanism is extensively used by the Eigen library so we get feedback from there, too.

Some classes have non-trivial invariants. For example, the dimensions of the sub-matrices in a `BlockMatrix` must satisfy certain conditions. For such classes, we add an ordinary private function `classInvariantsHold_` that returns `true` if and only if the invariants hold. Functions that perform operations that might affect the invariants (like constructors or assignment operators) end with

```
assert(this->classInvariantsHold_());
```

to check they did not cripple the object's state. If debugging is disabled, the above line produces no machine code.

Finally, we are using a number of macros that are strategically placed at points where something bad might happen.

- `BSC_KLAMMLER_NOT_REACHED` always and unconditionally throws an exception. It is placed in “unreachable” `default` cases of `switch` statements and the like that are expected to be never reached if our reasoning is correct. Using this macro is safer than simply placing a comment stating our expectation, more expressive than `assert(0)` and if used consistently, allows quicker understanding of the code.
- `BSC_KLAMMLER_NOT_IMPLEMENTED` is another macro that triggers a fatal exception. It is placed in functions that still need to be implemented. This is safer than simply returning a dummy value because we cannot forget to implement them properly this way. Also, it makes it easy to search the code for open issues.
- `BSC_KLAMMLER_INEFFICIENT_ALGORITHM` is placed inside algorithms that are already implemented (and supposed to work correctly) but known to be inefficient. The macro calls a diagnostic function that by default does

nothing but can be configured to report what inefficient algorithms are (still) called. See the discussion in section [4.2.2.1](#) for details about it.

We make extensive use of the “resource allocation is initialization” (RAII) pattern to keep our code clean and safe. For example, we are using smart pointers throughout our code to avoid having to deal with memory deallocations ourselves. Thanks to C++11’s `std::unique_ptr`, this comes at zero run-time overhead. All of our classes implement copy and move semantics and make the need for using pointers very small. RAII is also used in other contexts to reliably start and stop timers or to join threads.

Chapter 5

Experimental

We have run some benchmarks for our algorithms that are discussed in this chapter. For the algorithms we have implemented, we wanted to know

- how much execution time they have,
- how high a FLOP rate they achieve and
- how large the errors are.

The number of FLOPs was counted as described in section 4.4.

To measure execution time, the `init` and `solve` function members of the `Preconditioner` class accept as an optional parameter a pointer to a timer that is started and stopped immediately before and after the operations are started and finished respectively. As timers, we use the best (most accurate) available clock that provides a steady notion of time. This is done easily using the `std::chrono` library included in C++11.

```
/** @brief Best available clock. */  
using clock_type = typename std::conditional<  
    std::chrono::high_resolution_clock::is_steady,  
    std::chrono::high_resolution_clock,  
    std::chrono::steady_clock>::type;
```

Clearly, if we know the number of FLOPs and the execution time, we can compute the FLOP rate, given that we already know the number of CPUs.

Finally, the errors are measured by multiplying the obtained result \mathbf{x} with the coefficient matrix \mathbf{M} (which we keep a copy of) and subtracting it from the right-hand side \mathbf{b} . The norm of this residual vector is then compared to the norm of \mathbf{b} . This gives us the relative residual

$$r_{\text{rel}} := \frac{\|\mathbf{r}\|}{\|\mathbf{b}\|} \quad \text{with} \quad \mathbf{r} := \mathbf{M}\mathbf{x} - \mathbf{b} . \quad (5.1)$$

About the problem instance, we record the number of non-zero entries as well as the recursion depth and the range in which the sizes of the blocks \mathbf{A} and \mathbf{B} fall at the lowest level.

Our program can be configured to report all this information by emitting SQL INSERT statements on standard output. These can be redirected to a file and then loaded into a database for further processing. A SQLite database file with the data discussed in this chapter is available from the author.

5.1 Test Setup

Since we had no collection of real-world problems to test our algorithms with, we have included a random problem generator in our software. It takes as parameters the desired recursion depth of the coefficient matrix, a range from which to chose the size of the smallest dense blocks A and B along the main diagonal and whether the generated matrix should be symmetric and positive definite (we always set that last parameter true).

The generator proceeds by picking randomly sized (within the given range) dense blocks at the lowest level and then recursively assembles them to a matrix that looks as if it were obtained via nested dissection (§ 2.4.1).

Once this structure is built up. The dense blocks are filled with random numbers. We use the `setRandom` function from Eigen's matrix class for this. If the matrix is to be symmetric and positive definite, we make it diagonal dominant as this is a sufficient condition¹. First, we mirror the lower triangular part to the upper to make it symmetric. Then we sum up the absolute values of each row and assign the sum to the diagonal element².

Likewise, a set of random right-hand vectors with no further constraints is generated.

The same matrix is then decomposed with both algorithms and the same set of right-hand side vectors is solved for.

The parameters were chosen as large as possible where the most significant constraint was not to exhaust the computer's memory. The recursion depth was chosen between 4 and 10 and the block sizes at the lowest level between 500 and 10 000. In order to obtain a linear measure for the matrix size, we look at the square root of the non-zero entries. This is what the size of a dense matrix with the same number of non-zero entries would be.

5.2 Hardware

We had access to two server computers called ITI-120 and ITI-127 in the following³. The most important technical details about these machines are summarized

¹This procedure was recommended to us by Daniel Maurer, who has implemented an industry-strength block LL^T solver for distributed computing environments [15].

²This is done using visitors (§ 4.2.1.1.1).

³They are known to the world as `i10pc120.iti.kit.edu` and `i10pc127.iti.kit.edu` respectively. Actually, we had four machines since there are also `i10pc121.iti.kit.edu` and `i10pc128.iti.kit.edu` which are of the same type. ITI stands for "Institute for Theoretical Informatics", our department at the Karlsruhe Institute of Technology (KIT).

Property	ITI-120	ITI-127
processor	Intel® Xeon® CPU E5345	Intel® Xeon® CPU E5-4640
architecture	x86_64	x86_64
CPUs ^a	8	32
AVX ^b	no	yes
CPU frequency	2.3 GHz	2.4 GHz
Cache	4 MiB	20 MiB
RAM	16 GiB	500 GiB
operating system	Ubuntu 10.04.4 LTS	Ubuntu 12.04.5 LTS
kernel	2.6.32-66-generic	3.2.0-64-generic

^a actual hardware CPUs without hyper-threading
^b Advanced Vector Extensions

TABLE 5.1: Technical data about our test hardware.

in table 5.1.

All experiments were performed with exclusive access to the machine. Except from an SSH server and other elementary operating system tasks, there were no concurrent jobs executing on the machines.

Unless explicitly mentioned otherwise, the results discussed in this chapter refer to experiments on ITI-127 using all available 32 CPUs and level-3 parallelization (§ 4.3) enabled. Our program is compiled with NeSt (§ 2.3.5.4) for matrix inversion and linked to the MKL.

5.3 Decomposition

As expected from the theoretical discussion, the block LDL^T variant performs considerably more work than the block LL^T algorithm. Although we have seen increased FLOP rates (as expected) especially with large inputs and many processors, the amount of additional work is so large (very roughly doubled) that it is extremely hard for the block LDL^T decomposition to compete against the LL^T version. Nevertheless, we have seen instances where the overall execution time was lower.

Figure 5.1 shows the theoretic number of FLOPs required for both algorithms for the test instances we have run. From the relative plot, it can be readily seen that the block LDL^T decomposition requires roughly the doubled amount of work but the ratio varies greatly even for problems of similar size.⁴

Plotted in figure 5.2 is the execution time for both algorithms. The absolute values again vary greatly but a general trend for the difference to diminish with

⁴This is actually a hint that simply counting non-zero elements is not a very good measure. This can also be seen from the two “outlier” groups around 60 k and 90 k.

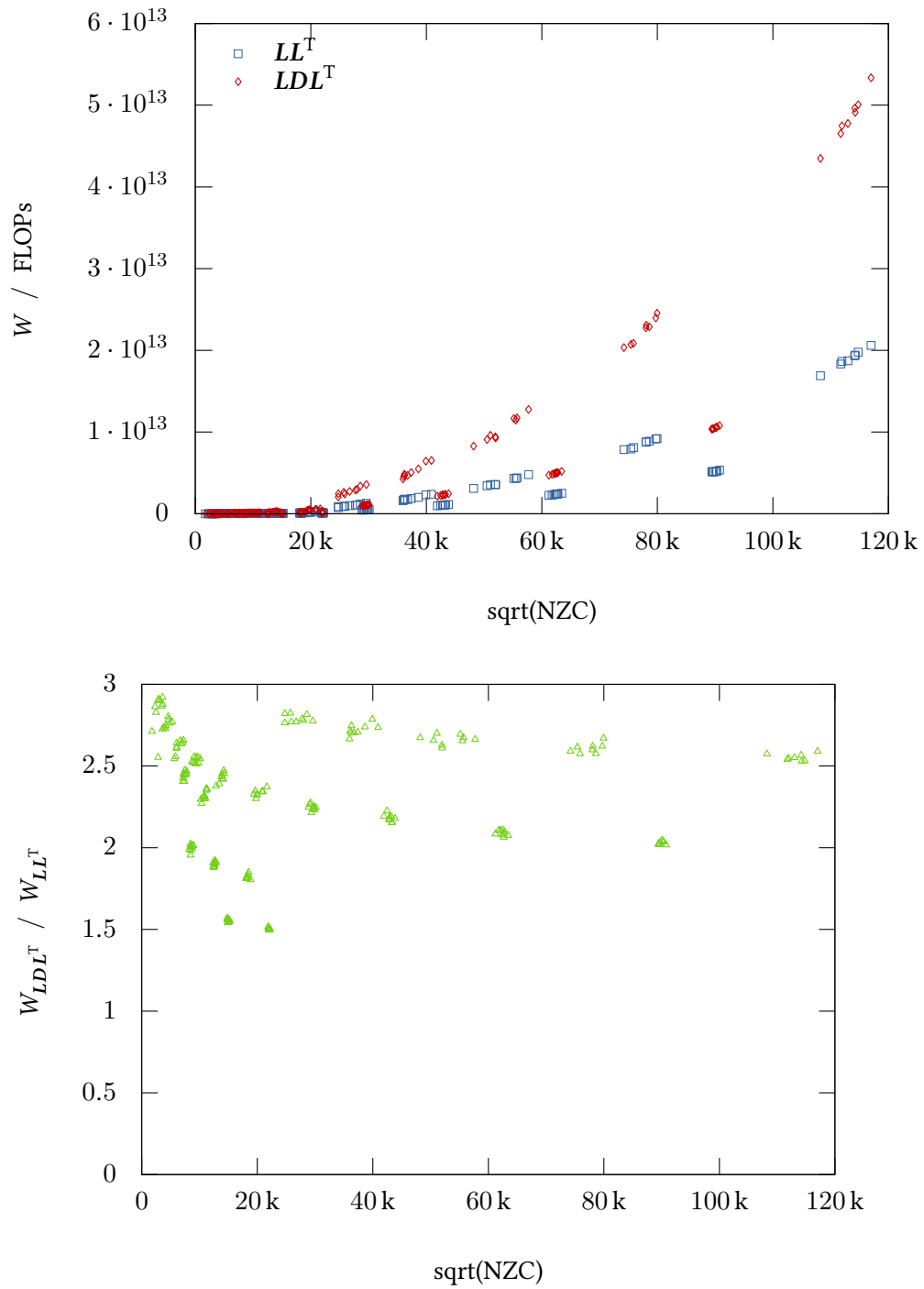


FIGURE 5.1: Absolute and relative amount of (theoretical) work for decomposing.

larger inputs can be seen. The relative plot reveals that while for small inputs ($\sqrt{\text{NZC}} \lesssim 20\text{k}$), the LDL^T variant is hopelessly outperformed by the LL^T version. For larger inputs, the ratio approaches 1 with the LDL^T variant being faster for about half of the instances.

Figure 5.3 compares the efficiency (FLOP rates) of the two algorithms. These show a clear result that we would have expected from our theoretical considerations. While the LL^T version performs many Cholesky decompositions and forward substitutions that have linear critical paths and require quite some branching, the LDL^T variant is dominated almost exclusively by matrix-matrix products that have logarithmic critical paths and can be implemented very efficiently, especially for large matrices (see § 4.1.2.5 and figures therein). For small inputs, both algorithms make poor use of the hardware and gain as the inputs grow. However, for the LL^T version, the FLOP rate levels off at about $1 \text{ GFLOP s}^{-1} \text{ CPU}^{-1}$ while the LDL^T variant easily reaches the doubled efficiency, thereby compensating for its about equally large additional work.

5.4 Solving

Unlike for the decomposition where the LDL^T variant pays a high price by performing much extra work, the cost for solving a linear system once the decomposition is computed is exactly the same for both algorithms. (The respective plot showing a straight line at a constant ratio of 1 is so boring that we refrain from even showing it.) However, the work for the block LL^T decomposition is in great parts dedicated to substitutions while the block LDL^T decomposition requires only matrix-vector products. With regard to efficiency, the same arguments as for the decomposition apply. This time however, we get the increased efficiency at no extra work.

Consequently, figure 5.4 shows that for almost any input, the LDL^T variant is significantly faster. On average, it only takes 60 % of the time to solve. The FLOP rate, as the amount of work is equal, is therefore increased accordingly. However, the factor of 2 or more seen for the decompositions is not reached.

5.5 Accuracy

We would have expected that computing the inverse matrix – especially with an inexact iterative algorithm – would have negative effects on the accuracy of the result. Plotting the logarithms of the relative residuals for the block LL^T and LDL^T decomposition against each other (fig 5.6) shows that they have a nice linear dependency with the LDL^T variant having a greater error as expected. To our surprise, linear regression reveals that the relative residuals for the LDL^T variant are less than 10 % greater on average. While this is significant, we assume that for most practical applications, such a small increase in the error will not matter and no further refinement of the result is needed.

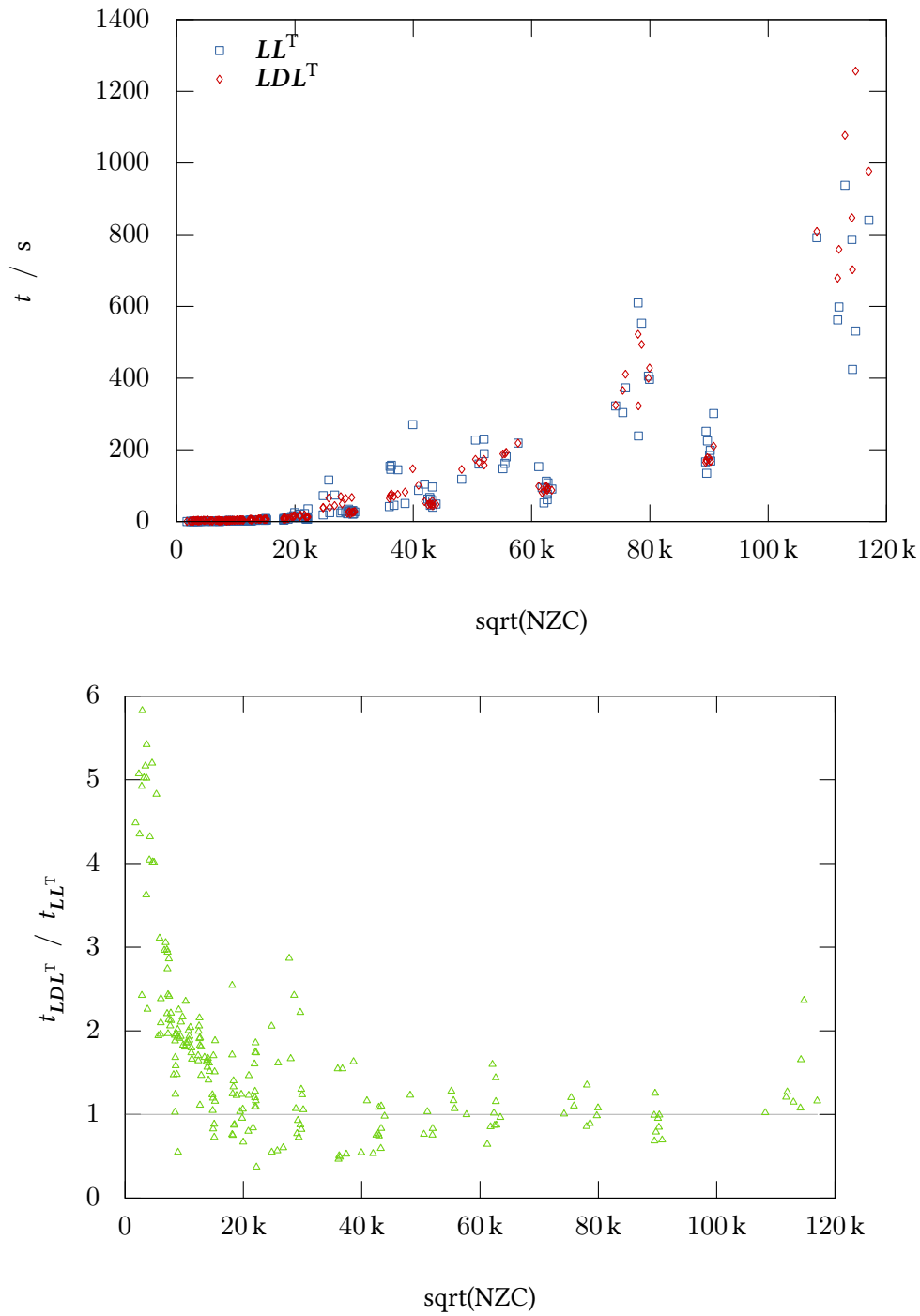


FIGURE 5.2: Absolute and relative execution times for decomposing.

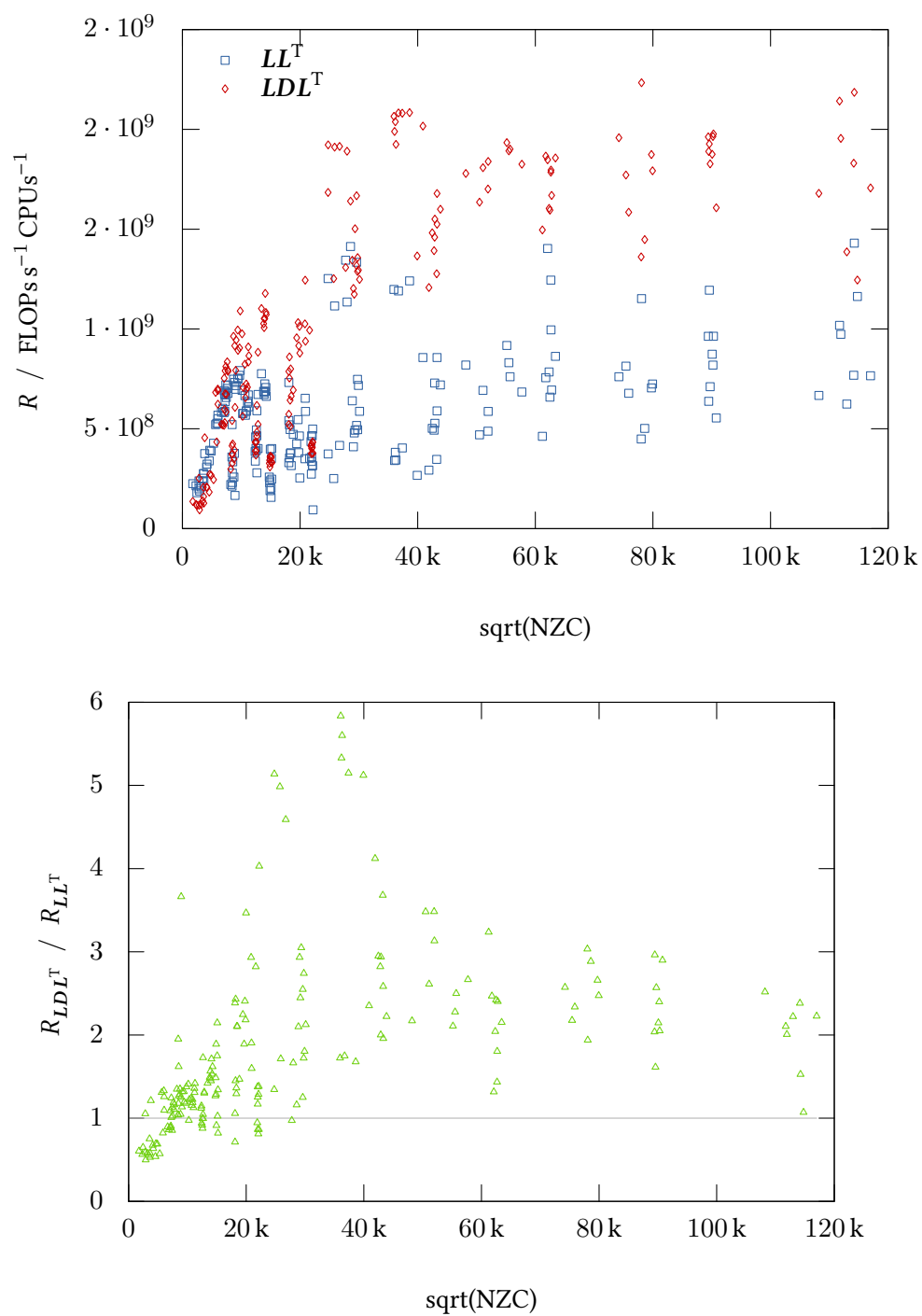


FIGURE 5.3: Absolute and relative FLOP rates for decomposing.

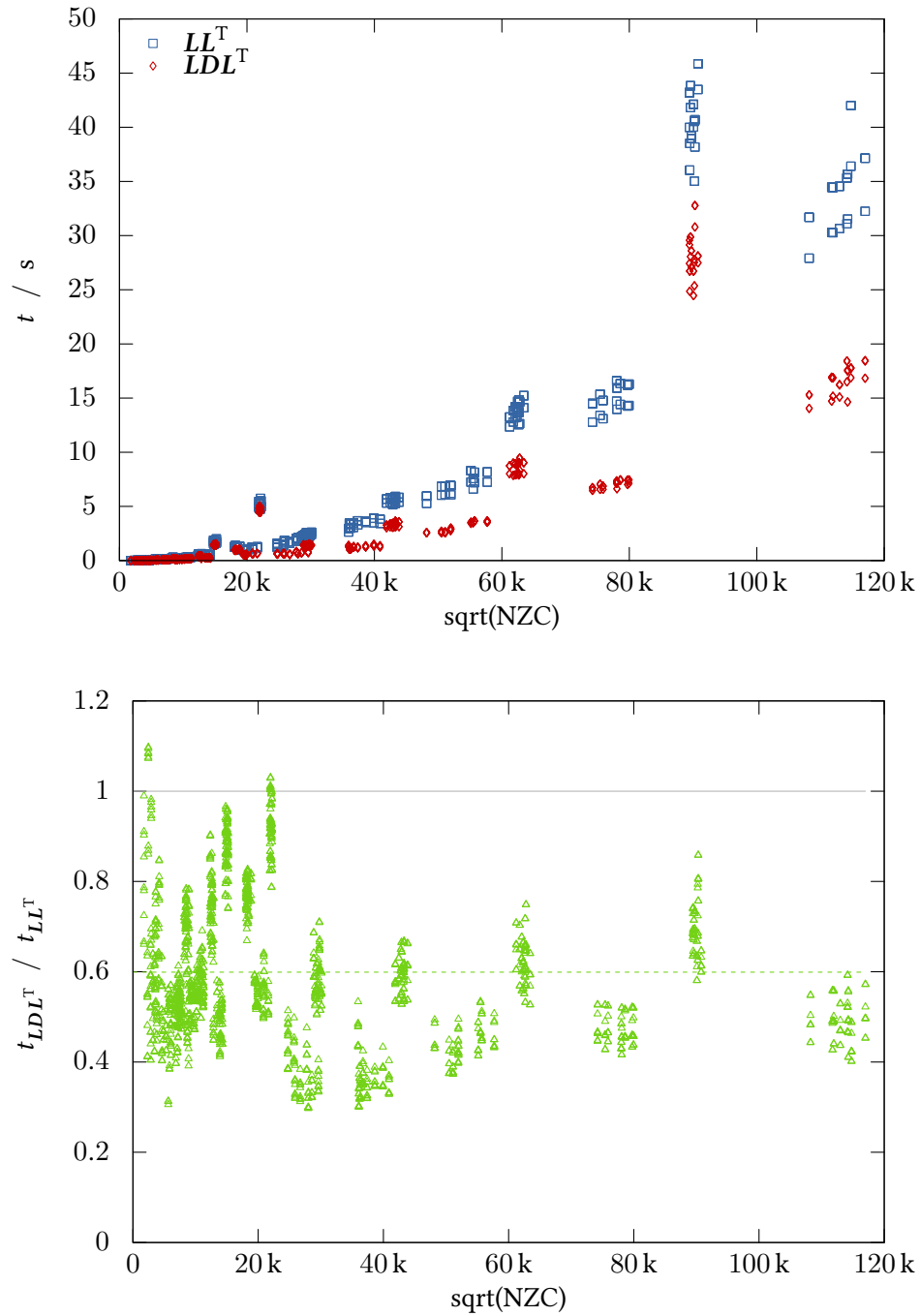


FIGURE 5.4: Absolute and relative execution times for solving. The dashed line is the average over all solves.

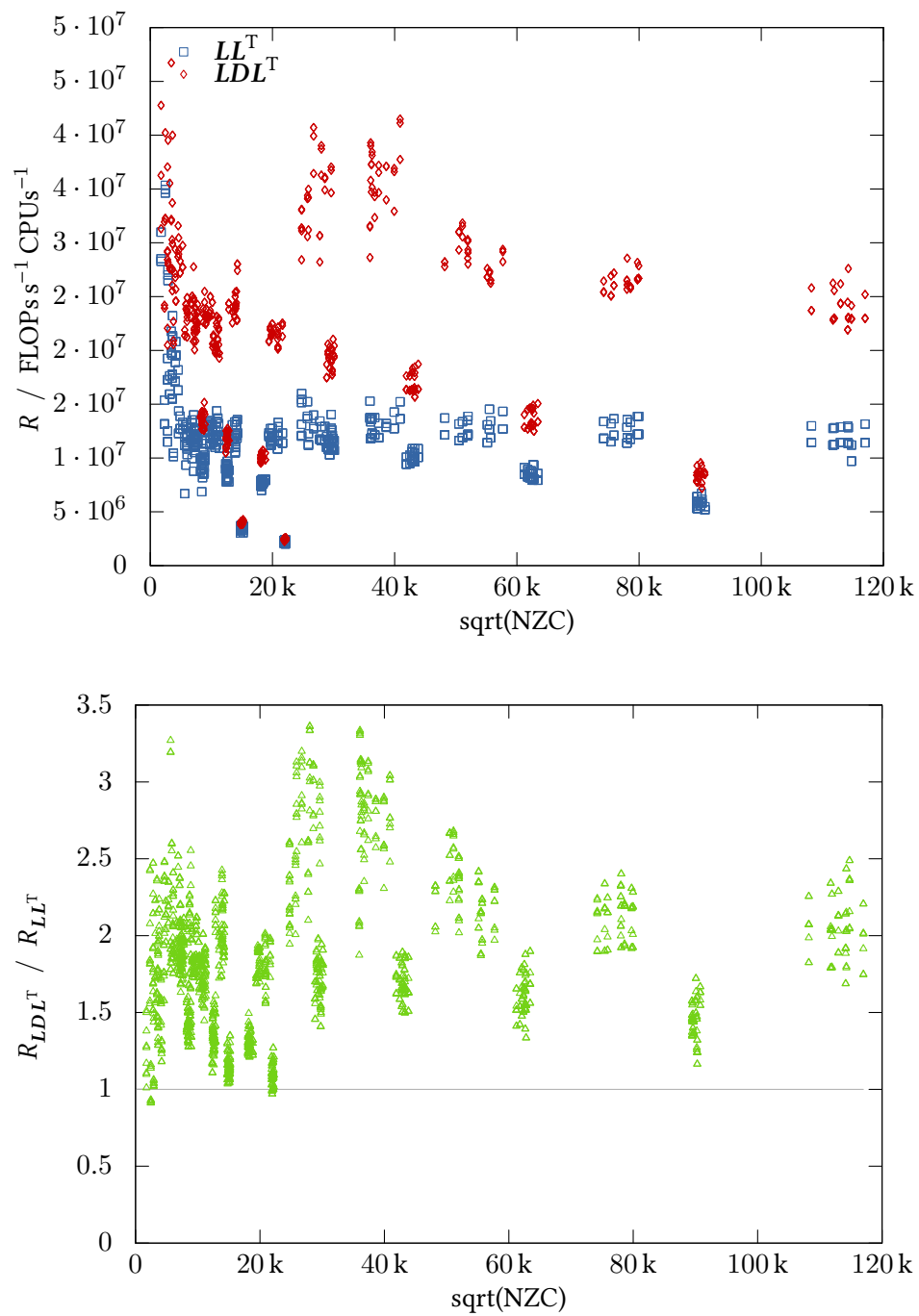


FIGURE 5.5: Absolute and relative FLOP rates for solving.

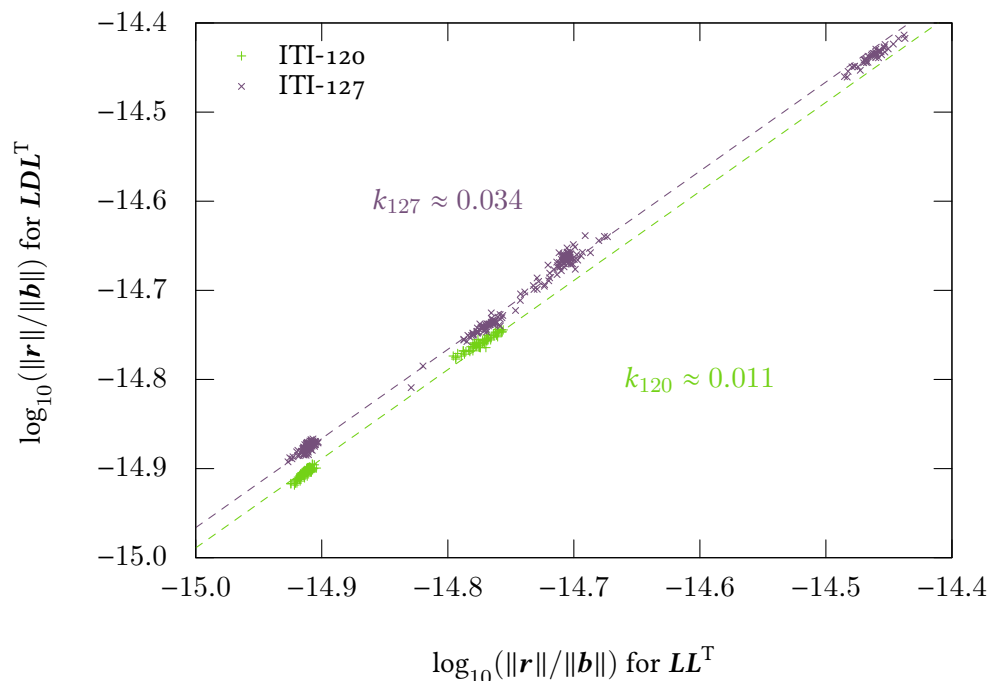


FIGURE 5.6: Logarithmic relative residuals of the block LL^T and LDL^T algorithm for the two machines we have tested. The linear regression curve with slope fixed at unity has a bias of $d_{120} \approx 0.011$ and $d_{127} \approx 0.034$ respectively which means that on average, the residuals for the LDL^T variant are about $10^{d_{120}} \approx 1.03$ and $10^{d_{127}} \approx 1.08$ times those of the LL^T version.

Note however that we only investigated very well conditioned symmetric and positive definite diagonal dominant coefficient matrices. Maybe for other matrices the results would have been not so good.

From the plot it can also be seen that matrices with larger blocks have larger absolute errors. Another interesting observation is that the relative errors are significantly larger on the ITI-127 machine. We don't really have an explanation for this except that maybe the AVX code has to do with it.

5.6 Tests on the Smaller ITI-120

On the smaller ITI-120 machine, we have been unable to produce satisfactory results. Both algorithms had a hard time approaching reasonable FLOP rates but even more so for the block LDL^T version. We think this is due to the inefficiency of parallel library functions for small matrices and the bookkeeping overhead that is more significant for small problems. As we have increased the problem size to a point where these negative effects might have been compensated for, we ran

out of memory. The plots we have obtained from the ITI-120 machine are not very interesting so we don't show any. (In essence, they show the same as the other diagrams but clipped to the region below 20 k.) The important observation is that our implementation requires a certain problem size to make good use of the hardware and that this limit is even higher for the block LDL^T decomposition.

Chapter 6

Conclusion

We have introduced a recursive block LL^T decomposition for symmetric and positive definite H-matrices with nested dissection structure and presented a variant thereof that is based on matrix inversion and produces a block LDL^T decomposition where the block diagonal matrix is obtained as its inverse. While the LL^T version spends a lot of work computing Cholesky factorization and forward substitutions, the LDL^T variant is dominated by matrix-matrix products. This is desirable because matrix products are extremely well understood and highly optimized implementations are available off-the-shelf. Furthermore, matrix-matrix multiplication only has a critical path logarithmic in the matrix size whereas Cholesky factorization and substitution both have linear critical paths that set limits to parallel computing. On the negative side, the LDL^T variant severely increased amount of work roughly by a factor of two. On the other hand, solving a linear system once the decomposition is computed is equally expensive for both versions but the LDL^T variant has a shorter critical path.

In this work we have presented our implementation of both algorithms for shared-memory machines using the C++ programming language and the Eigen linear algebra library. We have conducted experiments on two server machines with 8 and 32 cores respectively. While for small inputs, the LDL^T variant is clearly not worthwhile, it amortizes as parallelism and problem size increase. We have hit instances where the block LDL^T decomposition outperforms block LL^T decomposition but the advantage is neither reproducible nor significant. In contrast, solving is almost always significantly faster as expected from theory.

We conclude that block LDL^T decomposition might be worth consideration in cases where many linear systems need to be solved for a single coefficient matrix in a highly parallel system. This will be especially compelling if the right-hand sides are not all known a priori but become ready over time so an effective parallelization scheme cannot be built solely upon solving independent systems in parallel.

We envision that a possible application domain could be in near-time computation for finite-element problems as found in many engineering disciplines. Our

implementation for shared-memory systems has shown to eventually equalize the penalty of its increased amount of work on a machine with 32 processors and 500 GiB memory. Such machines might be found already today in engineering offices and even more so in the near future.

6.1 Further Work

There are several directions this work could be extended to. Maybe the most obvious step could be to port our implementation to a distributed memory system where even larger inputs could be tested.

In addition, we have thought of the possibility to make a hybrid algorithm between block LL^T and LDL^T by only inverting some of the matrices (those near the root of the dependency tree where most parallelism is needed) and computing Cholesky factorizations for the others. This could allow tuning the algorithm between less work and more efficiency.

$$\begin{matrix}
 \begin{matrix} \text{[Blue Block Matrix]} \\ M \end{matrix} & = & \begin{matrix} \text{[Green Block Matrix]} \\ L \end{matrix} & \cdot & \begin{matrix} \text{[Red Diagonal Matrix]} \\ D \end{matrix} & \cdot & \begin{matrix} \text{[Green Block Matrix]} \\ L^T \end{matrix} \\
 & & & & & & (6.1)
 \end{matrix}$$

On the other hand, doing so might destroy the advantage we currently see for solving.

Bibliography

- [1] Noga Alon, Amir Shpilka, and Christopher Umans. “On sunflowers and matrix multiplication”. In: *Computational Complexity* 22.2 (2013), pp. 219–243. ISSN: 1016-3328. DOI: 10.1007/s00037-013-0060-1.
- [2] *Basic Linear Algebra Subprograms (BLAS)*. URL: <http://www.netlib.org/blas/> (visited on 11/04/2014).
- [3] *Boost C++ Libraries*. URL: <http://www.boost.org/> (visited on 11/04/2014).
- [4] Steffen Börm, Lars Lars Grasedyck, and Wolfgang Hackbusch. *Hierarchical Matrices*. lecture note 21. Max-Planck-Institut für Mathematik in den Naturwissenschaften Leipzig, June 2006.
- [5] Erich Gamma et al. *Design Patterns : Elements of Reusable Object-Oriented Software*. Boston, USA: Addison-Wesley, 2007.
- [6] Alan George, Michael T. Heath, and Joseph Liu. “Parallel Cholesky factorization on a shared-memory multiprocessor”. In: *Linear Algebra and its Applications* 77.0 (1986), pp. 165–187. ISSN: 0024-3795. DOI: [http://dx.doi.org/10.1016/0024-3795\(86\)90167-9](http://dx.doi.org/10.1016/0024-3795(86)90167-9).
- [7] Gene H Golub and Charles F van Loan. *Matrix Computations*. Oxford, Great Britain: North Oxford Academic, 1986. ISBN: 0-946536-00-7; 0-946536-05-8.
- [8] Raymond Greenlaw, H James Hoover, and Walter L Ruzzo. *Limits to Parallel Computation: P-completeness Theory*. New York, USA: Oxford University Press, Inc, 1995. ISBN: 0-19-508591-4.
- [9] Andrew Hunt and David Thomas. *The pragmatic programmer : from journeyman to master*. Boston, USA: Addison-Wesley, 1999. ISBN: 0-201-61622-X.
- [10] Benoît Jacob, Gaël Guennebaud, et al. *Eigen*. URL: <http://eigen.tuxfamily.org/> (visited on 11/04/2014).
- [11] Vipin Kumar. *How to use Boost uBLAS with Intel MKL?* Apr. 24, 2013. URL: <https://software.intel.com/en-us/articles/how-to-use-boost-ublas-with-intel-mkl> (visited on 11/04/2014).
- [12] François Le Gall. “Powers of Tensors and Fast Matrix Multiplication”. In: *CoRR abs/1401.7714* (2014). URL: <http://arxiv.org/abs/1401.7714>.

- [13] *Linear Algebra Package (LAPACK)*. URL: <http://www.netlib.org/lapack/> (visited on 11/04/2014).
- [14] *Math Kernel Library (MKL)*. Intel Corporation. URL: <https://software.intel.com/en-us/intel-mkl> (visited on 11/04/2014).
- [15] Daniel Maurer and Christian Wieners. “A parallel block *LU* decomposition method for distributed finite element matrices”. In: *Parallel Computing* 37.12 (2011). 6th International Workshop on Parallel Matrix Algorithms and Applications (PMAA’10), pp. 742–758. ISSN: 0167-8191. DOI: <http://dx.doi.org/10.1016/j.parco.2011.05.007>.
- [16] Victor Pan and John Reif. “Efficient Parallel Solution of Linear Systems”. In: *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*. STOC ’85. Providence, Rhode Island, USA: ACM, 1985, pp. 143–152. ISBN: 0-89791-151-2. DOI: [10.1145/22145.22161](https://doi.org/10.1145/22145.22161).
- [17] Victor Pan and John Reif. “Fast and Efficient Parallel Solution of Sparse Linear Systems”. In: *SIAM Journal on Computing* 22.6 (Dec. 1993), pp. 1227–1250. ISSN: 0097-5397. DOI: [10.1137/0222073](https://doi.org/10.1137/0222073).
- [18] Roldan Pozo. *Template Numerical Toolkit (TNT)*. National Institute of Standards and Technology. URL: <http://math.nist.gov/tnt/> (visited on 11/04/2014).
- [19] William H Press et al. *Numerical Recipes: The Art of Scientific Computing*. 3rd ed. Cambridge, Great Britain: Cambridge University Press, 2007.
- [20] Peter Sanders, Jochen Speck, and Raoul Steffen. “Work-efficient Matrix Inversion in Polylogarithmic Time”. In: *Proceedings of the Twenty-fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA ’13. Montreal, Quebec, Canada: ACM, 2013, pp. 214–221. ISBN: 978-1-4503-1572-2. DOI: [10.1145/2486159.2486173](https://doi.org/10.1145/2486159.2486173).
- [21] Volker Strassen. “Gaussian elimination is not optimal”. In: *Numerische Mathematik* 13.4 (1969), pp. 354–356. ISSN: 0029-599X. DOI: [10.1007/BF02165411](https://doi.org/10.1007/BF02165411).
- [22] Bjarne Stroustrup. *The C++ Programming Language*. 4th ed. Boston, USA: Addison-Wesley, 2014.
- [23] Joerg Walter, Mathias Koch, et al. *Basic Linear Algebra Library*. Ed. by David Bellot. URL: http://beta.boost.org/doc/libs/1_56_0/libs/numeric/ublas/doc/index.htm (visited on 11/04/2014).

List of Algorithms

2.1	Dot	9
2.2	ForwardSubstitution	13
2.3	BackwardSubstitution	14
2.4	CholeskyInnerProduct	16
2.5	CholeskyGaxpy	18
2.6	InverseNewton	24
2.7	InverseNeSt	26
3.1	NDBlockLLT	38
3.2	NDBlockLLTLeaf	39
3.3	NDBlockLDLTLeaf	45

List of Code Listings

4.1	CBLAS Interface Example	51
4.2	Boost uBLAS Example	52
4.3	TNT Example	54
4.4	Eigen Example	54
4.5	naïve Self-Made Matrix Multiplication Code	56

List of Figures

2.1	H-matrix as Quad-Tree	28
2.2	Nested Dissection	31
3.1	Data Dependencies for Block LL^T Decomposition	36
3.2	Call Tree for Block LL^T Decomposition	41
4.1	FLOP Rates for Linear Algebra Libraries – Sequential	57
4.2	FLOP Rates for Linear Algebra Libraries – Parallel	58
4.3	Composite Pattern	60
4.4	HMatrix Class Diagram	66
4.5	Product of HMatrix and HVector	70
4.6	Products of ArrangedMatrixes	72
4.7	Preconditioner Class Diagram	74
4.8	Parallelization Scheme	76
5.1	Work for Decomposing	83
5.2	Execution Times for Decomposing	85
5.3	FLOP Rates for Decomposing	86
5.4	Execution Time for Solving	87
5.5	FLOP Rate for Solving	88
5.6	Relative Residuals	89

List of Tables

2.1	Blocks of Nested Dissection Matrix	30
3.1	Work for Block LL^T Decomposition	40
3.2	Work for Block LDL^T Decomposition	46
4.1	Linear Algebra Library Comparison	55
4.2	Special ArrangedMatrixes	64
5.1	Test Hardware Technical Data	82

