Diplomarbeit

Exascale Ready Work-Optimal Matrix Inversion

Raoul Steffen

R-Steffen@gmx.de

Advisors:
Jochen Speck
Prof. Dr. Peter Sanders

14th August 2012

Zusammenfassung auf Deutsch

In dieser Diplomarbeit entwickle ich einen neuen Algorithmus OPT zur Inversion von Matrizen. Ich beweise Eigenschaften zur parallelen Laufzeit und zum Arbeitsaufwand von OPT. OPT ist kombiniert aus Strassens Algorithmus zur Inversion von Matrizen und aus Newton Approximation und basiert auf einer Subroutine zur Matrixmultiplikation.

OPT ist ein arbeitsoptimaler Algorithmus, d.h. er benötigt höchstens einen konstanten Faktor mehr Arbeit als jeder andere (arbeitsoptimale) Algorithmus. Außerdem benötigt OPT nur polylogarithmische Zeit auf höchstens $O(n^3)$ Prozessoren, wobei die Prozessorzahl von der Multiplikationsroutine bestimmt wird. Damit vereint er diese beiden Vorteile von Strassens Algorithmus und Newton Approximation.

Ich beweise eine neue Abschätzung zur numerischen Stabilität von Strassens Algorithmus kombiniert mit Newton Approximation.

Im Zuge der Diplomarbeit habe ich OPT, Strassens Inversionsalgorithmus und Newton Approximation zusammen mit einer Matrixcontainerklasse in einem flexiblen Testprogramm implementiert. Ich beschreibe das Design der Implementierung und die Verwendung und Schwierigkeiten von BLAS für die Matrixmultiplikationsroutine.

Im experimentellen Teil vergleiche ich die Laufzeit und die numerische Stabilität von OPT mit der Routine aus der Intel Math Kernel Library (MKL). Die konstanten Faktoren des Arbeitsaufwands von OPT erweisen sich als nicht mehr als doppelt so hoch wie die der MKL-Routine. Wie vorhergesagt skaliert OPT sehr gut. Selbst auf einem Computer mit nur acht Kernen ist er bereits deutlich schneller als die MKL-Routine.

Bezüglich der numerischen Stabilität werden OPT und Strassens Algorithmus dessen schlechtem Ruf nicht gerecht. Stattdessen erzeugen sie Ergebnisse vergleichbar mit denen der MKL-Routine. Ich entdecke eine unerwartete Instabilität von Newton Approximation wodurch sie schlechtere Ergebnisse erzeugt als alle anderen Algorithmen in der Implementierung. Zu dieser Instabilität präsentiere ich einige weitere Experimente.

Ich erkläre, die vorliegende Arbeit eigenständig und nur unter Verwendung der angegeben Hilfsmittel angefertigt zu haben.

Abstract

In this thesis I present a new algorithm OPT for matrix inversion that builds on a matrix multiplication subroutine. It is combined of Strassen's matrix inversion algorithm and Newton approximation. OPT overcomes the linear lower bound in parallel runtime of Strassen's inversion algorithm and traditional Gaussian elimination without the log-factor more work of Newton approximation. In particular I prove, that given a work-optimal multiplication subroutine that runs in polylog time, OPT not only runs in polylog time, too, but furthermore only needs a constant factor more work than any work-optimal inversion algorithm.

Additionally, I present a new stability result for Strassen's matrix inversion algorithm combined with Newton approximation.

As part of this thesis, I implemented OPT, Strassen's inversion algorithm, and Newton approximation in a flexible test program along with a matrix container class optimized for this purpose. I describe the design of this implementation and the use and difficultys of BLAS for the matrix multiplication subroutine.

In the experimental part, I compare the runtime of OPT to the routine included in the Intel Math Kernel Library (MKL) and observe its numerical stability. The constant factors on the amount of work of OPT show to be no larger than twice those of the MKL routine. As predicted, OPT shows to be very scaleable. Even on a computer with only eight cores it is already significantly faster than the MKL routine.

Concerning numerical stability, OPT and Strassen's algorithm do not live up to its bad reputation. Instead they produce results comparable to the MKL routine. I discover an unexpected instability of Newton approximation that makes it produce worse results than any other algorithm in the implementation. About this instability I present some further experiments.

Contents

List of Figures

1 Introduction

Gaussian elimination is well known as the basic matrix inversion algorithm. It uses partial pivoting (i.e. row interchanging) by the first non-zero element to ensure no division by zero occurs. It is considered to be numerically stable in practice when using partial pivoting by the absolute maximum element, although counterexamples can be constructed. Numerical stability has been proven for diagonally dominant and for positive definite matrices.[8] Gaussian elimination with partial pivoting is a practical algorithm for single core applications with the additional benefit of (almost) inplace operation.[1] Parallelization is only possible on a very low level, on $\mathrm{O}(n^2)$ elements in subtracting the pivot-row from all other rows and on $\mathrm{O}(n)$ elements in finding the maximum of a column and in division of the pivot-row by the pivot. Blocked versions exist that increase the number of elements for parallelization by a constant factor.

Strassen's matrix inversion algorithm recursively breaks down the inversion into matrix multiplications. Since most work is done in large matrix multiplications and matrix multiplications parallelize quite well, Strassen's algorithm offers more coarse grained parallelism and is better suited for large parallel systems. Furthermore, it can benefit from advanced multiplication algorithms. It recursively breaks down an inversion of size n into two dependent inversions of size $\frac{n}{2}$ and some multiplications.[2] The recursive inversions have to be done in serial; at the bottom of the recursion are n base inversions. Thus, Strassen's inversion algorithm has a critical path length in $\Theta(n)$. Also, it is infamous for poor numerical stability.

Newton approximation can be used to invert matrices, too. Each iteration squares the remaining error, while the initial error depends on the condition and the size of the matrix. With a suitable initial approximation, the number of necessary iterations to reach a constant error bound is in $\Theta(\log n)$.[12] While these iterations consist of full size matrix multiplications and thereby parallelize well, their number adds a factor of $\Theta(\log n)$ to the work-complexity. Since each iteration does not depend on an exact intermediate result of the previous iteration but only on an increasingly accurate approximation of the inverse, Newton approximation has the unique advantage of being able to correct computational errors.

The algorithm developed in this thesis combines Strassen's algorithm with Newton approximation to incorporate the advantages of both. It needs polylogaritmic parallel time while still doing only a constant factor times the work of a work-optimal inversion algorithm. It is as numerically stable as Newton approximation, while the experiments show even better results.

1.1 Previous Work in Parallel Matrix Inversion

With the arise of multi-core computers, research has been put into efficiently parallelizing linear algebra operations including matrix inversion.

In the practical field, blocked versions of well known algorithms such as Gaussian elimination have been established and are implemented in valued li-

[1] Only one additional column needs to be stored.

[2] In the general case, Strassen's inversion algorithm needs six multiplications of size $\frac{n}{2}$. For symmetric matrices only four multiplications a necessary, two of which yield a symmetric matrix.

braries such as ATLAS[18] and, most likely, Intel's MKL. Ongoing research tries to further refine them and improve their implementation. Recently, the technique of task scheduling has become popular. It offers greater flexibility than fixed distribution of the workload and loosens the intrinsic sequentiality of the greater parts of the algorithms. The tasks are generated from blocked algorithms as operations on the blocks. An overview over a class of such algorithms used in Netlib's LAPACK[17] has been given by Buttari, Langou, Kurzak, and Dongarra.[4] Song, YarKahn, and Dongarra and Kurzak, Ltaief, Dongarra, and Badia published about decentral scheduling of the tasks.[2, 3] However, all blocked algorithms sustain a critical path length in $\Omega(n)$.

In the theoretical field, research tries for algorithms that satisfy polylog time bounds. In 1974, Csanky published the first algorithm to break the asymptotic bound of $\Omega(n)$ but only by $\mathrm{O}(\log^2 n)$, i.e. his algorithm took $2n - \mathrm{O}(\log^2 n)$ parallel steps, proving algorithms known before to be not optimal.[14] In 1976, he published the first algorithm that works in polylog time, proving it possible.[13] Many algorithms were published that are quite complicated and mostly work with computing the characteristic polynomial as a critical intermediate step. Additionally they usually require exact computation over a finite field or over rational numbers.

In 1994, Reif proposed a recursive polylog time algorithm that directly computes the inverse or, in variants, LU or QR decompositions.[9] However, it remains unclear from his publication how the critical path can be shorter than n without the help of other polylog inversion algorithms.

The only widely accepted algorithm of practical usability that works with floating point represented real numbers and also satisfies polylog times bounds is Newton approximation. It requires a suitable initial approximation, though, of which was unknown how to compute it efficiently (i.e. without inverting the matrix) until the publication of Pan and Reif 1985.[10, 12] Codenotti, Leoncini, and Preparata proposed Newton approximation to be the only known polylog time inversion algorithm admitting numerically stable implementations.[7] Contrary to that, by the measures and calculations of Demmel, Dumitrui, and Holtz, generally all linear algebra operations can be done in polylog time as stable as they can be done in serial execution, but not necessarily with the same work-complexity.[5]

1.2 Notation

Throughout this thesis n shall denote the sidelength of the input matrix.

Almost all of the logarithms in this thesis are of base 2. Therefore, if not denoted otherwise, $\log = \log_2 = \mathrm{ld}$.

When analyzing the work-complexity and parallel runtime of an algorithm A, $W_A(n)$ and $T_A(n)$ shall denote the number of operations in the RAM model and the number of timesteps in the PRAM model, respectively. Because parallelism comes from matrix multiplications, additions, and reductions, the number of processors is determined by the matrix multiplication subroutine and is at most n^3. M and I stand for (hypothetical) optimal multiplication and inversion algorithms.

2 Known Algorithms

In this section I describe the two algorithms OPT is based on. I analyze their work, parallel time and storage requirements. In sections 2.1.2 and 2.1.3 I present new insight about the stability of Strassen's inversion algorithm and the use of symmetric positive definite matrices.

In the analysis I need two lower bounds for the work of a matrix multiplication $W_M(n)$: $W_M(n) \geq n^2$ holds because the output alone has n^2 elements that have to be written. $W_M(n) \geq 4W_M(\frac{n}{2})$ is, on one hand, actually quite loose but, on the other hand, not known to be true in all cases.

For now I will call the factor γ, i.e. I discuss why I assume $\gamma = 4$ in $W_M(n) \geq \gamma W_M(\frac{n}{2})$. γ corresponds to the highest order in $W_A(n)$. When using (recursive) standard multiplication it is $\gamma = 8$. Strassen's inversion algorithm, neglecting lower order terms, reduces it to $\gamma = 7$. Other algorithms exist, for example Winograd's, that bring γ even closer to 4 at the cost of even more work hidden in lower order terms. The asymptotically fastest known algorithm reaches $\gamma = 5.179$ in average, i.e. requires $\mathrm{O}(n^{2.3727})$ operations.[1] It is believed by some specialists, that there might be a family of inversion schemes which asymptotically converges to $\gamma = 4$. However, there can be no algorithm that gives $\gamma < 4$ for almost all n, because it would then violate the first bound.

2.1 Strassen's Matrix Inversion Algorithm

In 1969 Volker Strassen published a recursive algorithm for matrix inversion.[15] The algorithm breaks down the inversion into several matrix multiplications. Thereby, Strassen showed that matrix inversion is not harder than matrix multiplication (see theorem 2.7). In the same paper, Strassen published his well known algorithm for matrix multiplication that is asymptotically faster than the classical multiplication algorithm and Gaussian elimination, showing that those algorithms are not optimal.

Breaking down the inversion into multiplications brings another benefit: In contrast to Gaussian elimination, matrix multiplications parallelize quite well.

2.1.1 Deriving Strassen's Inversion Algorithm from Gaussian Elimination

Let M be a matrix to invert, partitioned into four submatrices with square A and B. Interpret it as 2×2 matrix whose elements are also matrices.

$$M = \begin{pmatrix} A & E \\ C & B \end{pmatrix}$$

Assuming all necessary inverses exist, invert it by Gaussian elimination:

$$\left(\begin{array}{cc|cc} A & E & 1 & 0 \\ C & B & 0 & 1 \end{array}\right)$$

$$\left(\begin{array}{cc|cc} 1 & A^{-1}E & A^{-1} & 0 \\ C & B & 0 & 1 \end{array}\right)$$

$$\left(\begin{array}{cc|cc} 1 & A^{-1}E & A^{-1} & 0 \\ 0 & B - CA^{-1}E & -CA^{-1} & 1 \end{array}\right)$$

input: $M = \begin{pmatrix} A & C^T \\ C & B \end{pmatrix}$

operation			type	operation count
R	$=$	$\text{Inv}(A)$	recursive inversion	$W_S(\frac{n}{2})$
$C\tilde{A}$	$=$	$C \cdot R$	multiplication	$W_M(\frac{n}{2})$
S	$=$	B	copy	$(\frac{n}{2})^2$
S	$-=$	$C\tilde{A} \cdot C^T$	multiplication	$W_M(\frac{n}{2})$
$\tilde{S}$	$=$	$\text{Inv}(S)$	recursive inversion	$W_S(\frac{n}{2})$
P	$=$	$\tilde{S} \cdot C\tilde{A}$	multiplication	$W_M(\frac{n}{2})$
P'	$=$	P^T	transposition	$(\frac{n}{2})^2$
R	$-=$	$(C\tilde{A})^T \cdot P$	multiplication	$W_M(\frac{n}{2})$

output: $\tilde{M} = \begin{pmatrix} R & P' \\ P & \tilde{S} \end{pmatrix}$

Figure 1: Pseudocode for Strassen's matrix inversion algorithm for symmetric matrices and operation count for each sub-operation.

$$S = B - CA^{-1}E$$

$$\left(\begin{array}{cc|cc} 1 & A^{-1}E & A^{-1} & 0 \\ 0 & 1 & -S^{-1}CA^{-1} & S^{-1} \end{array}\right)$$

$$\left(\begin{array}{cc|cc} 1 & 0 & A^{-1} + A^{-1}ES^{-1}CA^{-1} & -A^{-1}ES^{-1} \\ 0 & 1 & -S^{-1}CA^{-1} & S^{-1} \end{array}\right)$$

Thus, if A and S are non-singular:

$$M^{-1} = \begin{pmatrix} A^{-1} + A^{-1}ES^{-1}CA^{-1} & -A^{-1}ES^{-1} \\ -S^{-1}CA^{-1} & S^{-1} \end{pmatrix}$$
$$\text{with } S = B - CA^{-1}E$$ [3]

For symmetric matrices $E = C^T$. Thus, if $A = A^T$:

$$M^{-1} = \begin{pmatrix} A^{-1} + A^{-1}C^TS^{-1}CA^{-1} & -A^{-1}C^TS^{-1} \\ -S^{-1}CA^{-1} & S^{-1} \end{pmatrix}$$
$$\text{with } S = B - CA^{-1}C^T$$

From the second formula follow the pseudocode shown in figure 1 and the DAG shown in figure 2.

Remark Gaussian elimination is the special case of Strassen's formula where A is chosen to be a 1×1 matrix.

[3]S is called the Schur complement of A.

2.1.2 Infeasibility of Pivoting

For general matrices, M being non-singular does not imply A to be non-singular, e.g.

$$M = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

is non-singular, but $A = 0$ is not. Gaussian elimination uses row-interchanges to overcome this problem. This process of swapping the current with a better suited row is called partial pivoting.[4] Gaussian elimination with partial pivoting by an absolute maximum element is considered to be numerically stable in practice, although there are counterexamples.[8, §3.4.6]

Baley and Ferguson proposed a non-functional method for pivoting for Strassen's inversion algorithm.[11] They mistakenly claimed that for any well conditioned matrix M with a half-sized partitioning at least one of the left partitions has to be well conditioned, too. That the claim does not hold can be easily seen at the following counterexample.

$$M = \left(\begin{array}{cc|cc} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ \hline 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{array}\right)$$

While M is a permutation matrix and thus has condition 1, none of the partitions is even non-singular.

Though pivoting is still possible, it is more complicated and does not guarantee low conditions.

Theorem 2.1 (Partial Pivoting)**.**
Given a non-singular matrix $M \in \mathbb{R}^{n\times n}$ and $k \in \{1,\dots,n\}$[5].
Then there is a subset of k rows of M so that their first k columns form a non-singular matrix A'.

Proof.
Let M_L be the submatrix of M consisting of the left k columns.

M is non-singular.
$\Rightarrow$ All columns of M are linearly independent.
$\Rightarrow$ All columns of M_L are linearly independent.
$\Rightarrow$ $\text{rank}(M_L) = k$
$\Rightarrow$ M_L has k linearly independent rows; those form a non-singular matrix A'.

□

There may be only one such subset of rows, e.g. in a permutation matrix there are only k rows whose left k columns are non-zero. Additionally, partial pivoting does not guarantee low conditions as can be seen at the following

[4]Full pivoting includes swapping columns as well but is uncommon in practice.
[5]Typically $k \approx \frac{n}{2}$

counterexample.

$$M = \frac{1}{\sqrt{3}} \begin{pmatrix} \sqrt{3} & 0 & 0 & 0 \\ 0 & 1 & \sqrt{2} & 0 \\ 0 & 1 & -\sqrt{\frac{1}{2}} & \sqrt{\frac{3}{2}} \\ 0 & 1 & -\sqrt{\frac{1}{2}} & -\sqrt{\frac{3}{2}} \end{pmatrix} \qquad k = 2$$

M is orthogonal, thus $\forall x \in \mathbb{R}^4 : \|Mx\| = \|x\|$, thus $\operatorname{cond}(M) = 1$.
Yet, all possible row selections yield $A' = \left(\begin{smallmatrix} 1 & 0 \\ 0 & \frac{1}{\sqrt{3}} \end{smallmatrix}\right)$ and $\operatorname{cond}(A') = \sqrt{3}$.

Finally, it is unclear how selecting rows could be parallelized effectively. Any algorithm that selects one row after the other would inevitably have a critical path length of at least k.

2.1.3 An Alternative to Pivoting

Given a non-singular matrix M, $\bar{M} = M^T M$ is always symmetric positive definite (see lemma A.1). From the inverse of $\bar{M}$, the inverse of M can be computed as

$$M^{-1} = M^{-1} M^{-T} M^T = (M^T M)^{-1} M^T = \bar{M}^{-1} M^T \ . \tag{1}$$

With this scheme, inversion of general matrices can be reduced to inversion of symmetric positive definite matrices at the cost of two multiplications. The condition of $\bar{M}$ is bound by the square of the condition of M (see lemma A.2).

Inverting a symmetric positive definite matrix has benefits over inverting a general one:

Theorem 2.2.
Let $M = \left(\begin{smallmatrix} A & C^T \\ C & B \end{smallmatrix}\right) \in \mathbb{R}^{n \times n}$ be symmetric positive definite with square $A \in \mathbb{R}^{k \times k}$. Then A and $S = B - CA^{-1}C^T$ are symmetric positive definite, in particular non-singular.

Proof.
Symmetry of A and S follows from symmetry of M.

Let $\quad \bar{\mathbf{x}} \in \mathbb{R}^k \setminus 0$

$$\mathbf{x} = \begin{bmatrix} \bar{\mathbf{x}} \\ 0 \end{bmatrix} \in \mathbb{R}^n \setminus 0$$

Then $\quad \bar{\mathbf{x}}^T A \bar{\mathbf{x}} = \mathbf{x}^T M \mathbf{x} > 0$

Thus, A is positive definite.

Observe

$$M = \begin{pmatrix} A & C^T \\ C & B \end{pmatrix} = \begin{pmatrix} I & 0 \\ CA^{-1} & I \end{pmatrix} \begin{pmatrix} A & 0 \\ 0 & S \end{pmatrix} \underbrace{\begin{pmatrix} I & A^{-1}C^T \\ 0 & I \end{pmatrix}}_{=:J}$$

$$\Rightarrow \quad J^{-T} M J^{-1} = \begin{pmatrix} A & 0 \\ 0 & S \end{pmatrix}$$

Let $\quad \bar{\mathbf{y}} \in \mathbb{R}^{n-k} \setminus 0$

$$\mathbf{y} = \begin{bmatrix} 0 \\ \bar{\mathbf{y}} \end{bmatrix} \in \mathbb{R}^n \setminus 0$$

Then $\quad \bar{\mathbf{y}}^T S \bar{\mathbf{y}} = (\mathbf{y}^T J^{-T}) M \underbrace{(J^{-1}\mathbf{y})}_{\neq 0} > 0$

Thus, S is positive definite.

□

Theorem 2.3.
Let $M = \left(\begin{smallmatrix} A & C^T \\ C & B \end{smallmatrix}\right) \in \mathbb{R}^{n\times n}$ symmetric positive definite with square $A \in \mathbb{R}^{k\times k}$. Then $\operatorname{cond}(A) \le \operatorname{cond}(M)$.

The proof follows after two lemmata.

Lemma 2.4.

Let $\quad M \in \mathbb{R}^{n\times n}$ *non-singular*

Then $\quad \operatorname{cond}(M) = \|M\| \cdot \|M^{-1}\| = \max\limits_{\|\mathbf{x}\|=1} \|M\mathbf{x}\| \Big/ \min\limits_{\|\mathbf{y}\|=1} \|M\mathbf{y}\|$

Proof.

By definition $\quad \|M\| = \max\limits_{\|\mathbf{x}\|=1} \|M\mathbf{x}\|$

It remains to show that $\quad \|M^{-1}\| = \dfrac{1}{\min\limits_{\|\mathbf{y}\|=1} \|M\mathbf{y}\|}$

$$\begin{aligned}
& \|M^{-1}\| = a \\
\Leftrightarrow\quad & \max_{\|\bar{\mathbf{y}}\|=1} \|M^{-1}\bar{\mathbf{y}}\| = a \\
\Leftrightarrow\quad & \|M^{-1}\bar{\mathbf{y}}\| = a \cdot \|\bar{\mathbf{y}}\| \\
\overset{\bar{\mathbf{y}}=M\mathbf{y}}{\Leftrightarrow}\quad & \|\mathbf{y}\| = a \cdot \|M\mathbf{y}\| \\
\Leftrightarrow\quad & \frac{1}{a} \cdot \|\mathbf{y}\| = \|M\mathbf{y}\| \\
\Leftrightarrow\quad & \frac{1}{a} = \min_{\|\mathbf{y}\|=1} \|M\mathbf{y}\|
\end{aligned}$$

□

Lemma 2.5. *(without proof)*
Let $U \supseteq V \supseteq W$ be vector spaces with orthogonal projections

$$\begin{tikzcd}
\mathbf{x} \arrow[r, mapsto] & \bar{\mathbf{x}} \arrow[r, mapsto] & \mathbf{x}' \\
U \arrow[r, two heads] \arrow[rr, bend right, two heads] & V \arrow[r, two heads] & W
\end{tikzcd}.$$

Then the diagram commutes and $\|\mathbf{x}\| \ge \|\bar{\mathbf{x}}\| \ge \|\mathbf{x}'\|$.

Proof of Theorem 2.3.
According to lemma 2.4, it suffices to show

1. $\max\limits_{\|\bar{\mathbf{x}}\|=1} \|A\bar{\mathbf{x}}\| \le \max\limits_{\|\mathbf{x}\|=1} \|M\mathbf{x}\|$

2. $\min\limits_{\|\bar{\mathbf{y}}\|=1} \|A\bar{\mathbf{y}}\| \geq \min\limits_{\|\mathbf{y}\|=1} \|M\mathbf{y}\|$

1.

$$\forall \bar{\mathbf{x}} \in \mathbb{R}^k : \|A\bar{\mathbf{x}}\| \leq \left\| \begin{bmatrix} A\bar{\mathbf{x}} \\ C\bar{\mathbf{x}} \end{bmatrix} \right\| = \left\| M \begin{bmatrix} \bar{\mathbf{x}} \\ 0 \end{bmatrix} \right\| \leq \max_{\|\mathbf{x}\|=1} \|M\mathbf{x}\|$$

2. Let $\bar{\mathbf{y}} \in \mathbb{R}^k$ with $\|\bar{\mathbf{y}}\| = 1$
   Let $\mathbf{y} = \begin{bmatrix} \bar{\mathbf{y}} \\ 0 \end{bmatrix} \in \mathbb{R}^n$
   It suffices to show that $\|A\bar{\mathbf{y}}\| \geq \min\limits_{\|\mathbf{y}'\|=1} \|M\mathbf{y}'\|$.
   The following form a chain of vector spaces as in lemma 2.5:

$$\begin{array}{ccccc} \mathbb{R}^n & \to & \mathbb{R}^k & \to & \langle \mathbf{y} \rangle^{[6]} \\ \mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} & \mapsto & \begin{bmatrix} x_1 \\ \vdots \\ x_k \end{bmatrix} & \mapsto & \langle \mathbf{x}, \mathbf{y} \rangle \cdot \mathbf{y}^{[7]} \\ M\mathbf{y} & \mapsto & A\bar{\mathbf{y}} & \mapsto & \langle M\mathbf{y}, \mathbf{y} \rangle \cdot \mathbf{y} \end{array}$$

Thus $$\|A\bar{\mathbf{y}}\| \geq \|\langle M\mathbf{y}, \mathbf{y} \rangle \cdot \mathbf{y}\| = \langle M\mathbf{y}, \mathbf{y} \rangle$$

Since M is symmetric positive definite, it can be diagonalized with respect to an orthonormal base:

$$M = QDQ^T$$
$$\text{with } Q^TQ = I \text{ and } D = \begin{pmatrix} d_1 & & 0 \\ & \ddots & \\ 0 & & d_n \end{pmatrix}$$

Write $\mathbf{y}$ in that base:

$$\mathbf{z} = Q^T\mathbf{y}$$

Let $$d' = \min_{\|\mathbf{y}'\|=1} \|M\mathbf{y}'\|$$

Then
$$\begin{aligned} \langle M\mathbf{y}, \mathbf{y} \rangle &= \langle QDQ^TQ\mathbf{z}, Q\mathbf{z} \rangle = \langle QD\mathbf{z}, Q\mathbf{z} \rangle = \langle D\mathbf{z}, \mathbf{z} \rangle \\ &= \left\langle \begin{bmatrix} d_1 z_1 \\ \vdots \\ d_n z_n \end{bmatrix}, \begin{bmatrix} z_1 \\ \vdots \\ z_n \end{bmatrix} \right\rangle \\ &= d'\|\mathbf{z}\|^2 + \left\langle \begin{bmatrix} (d_1 - d')z_1 \\ \vdots \\ (d_n - d')z_n \end{bmatrix}, \begin{bmatrix} z_1 \\ \vdots \\ z_n \end{bmatrix} \right\rangle \\ &\geq d' = \min_{\|\mathbf{y}'\|=1} \|M\mathbf{y}'\| \end{aligned}$$

□

[6] $\langle \mathbf{y} \rangle$ denotes the subspace spanned by $\mathbf{y}$, i.e. $\langle \mathbf{y} \rangle = \{a\mathbf{y} \mid a \in \mathbb{R}\}$.
[7] $\langle \cdot, \cdot \rangle$ denotes the scalar product inducing $\| \cdot \|$.

Theorem 2.6.
Let $M = \left(\begin{smallmatrix} A & C^T \\ C & B \end{smallmatrix}\right) \in \mathbb{R}^{n\times n}$ *be symmetric positive definite with square* $A \in \mathbb{R}^{k\times k}$ *and* $S = B - CA^{-1}C^T$*. Then* $\mathrm{cond}(S) \le \mathrm{cond}(M)$*.*

Proof.
Remember

$$M^{-1} = \begin{pmatrix} * & * \\ * & S^{-1} \end{pmatrix}$$

is symmetric positive definite.
Therefore by theorem 2.3:

$$cond(S) = cond(S^{-1}) \le cond(M^{-1}) = cond(M)$$

□

The algorithm proposed in this thesis works with symmetric positive definite matrices. For general non-singular matrices I assume conversion via the scheme (1). Therefore, for the remainder of this thesis, I focus on inversion of symmetric positive definite matrices.

2.1.4 Work

From the pseudocode shown in figure 1, I easily get a recurrence for the operation count.

$$\begin{aligned} W_S(n) &= 2W_S\left(\frac{n}{2}\right) + 4W_M\left(\frac{n}{2}\right) + 2\left(\frac{n}{2}\right)^2 \\ W_S(1) &= 1 \end{aligned}$$

Remark On current computers and matrix multiplication implementations, a matrix multiplication may include constant factors and transpositions without a penalty (see section 4.1 for details). For that reason, I do not count those operations separately.

The recurrence solves to:

$$\begin{aligned} W_S(n) &= 2\left(W_S\left(\frac{n}{2}\right) + 2W_M\left(\frac{n}{2}\right) + \left(\frac{n}{2}\right)^2\right) \\ &= nW_S(1) + \sum_{i=1}^{\log n} 2^i \left(2W_M\left(\frac{n}{2^i}\right) + \left(\frac{n}{2^i}\right)^2\right) \\ &\overset{W_M(\frac{n}{2}) \le \frac{1}{4}W_M(n)}{\le} nW_S(1) + \sum_{i=1}^{\log n} 2^i \left(\frac{2}{4^i}W_M(n) + \frac{n^2}{4^i}\right) \\ &= nW_S(1) + \sum_{i=1}^{\log n} \frac{2}{2^i}W_M(n) + \frac{n^2}{2^i} \\ &= nW_S(1) + \left(W_M(n) + \frac{n^2}{2}\right) \sum_{i=1}^{\log n} \left(\frac{1}{2}\right)^{i-1} \\ &= nW_S(1) + \left(W_M(n) + \frac{n^2}{2}\right) \frac{1 - \left(\frac{1}{2}\right)^{\log n}}{1 - \frac{1}{2}} \end{aligned}$$

$$\le nW_S(1) + \left(W_M(n) + \frac{n^2}{2}\right)\frac{1}{1-\frac{1}{2}}$$
$$= nW_S(1) + \left(2W_M(n) + n^2\right)$$
$$\overset{n^2 \le W_M(n)}{\Rightarrow} \quad W_S(n) \le 4W_M(n) \tag{2}$$

It follows

Theorem 2.7.
Matrix multiplication and matrix inversion are equally hard.

By this theorem in combination with his matrix multiplication algorithm, Strassen has shown that inversion by Gaussian elimination is not optimal.

Proof of Theorem 2.7.
Let $W_I(n)$ be the number of operations done by an optimal matrix inversion algorithm to invert a $n \times n$ matrix. From inequation (2) follows $W_I(n) \le W_S(n) \in \mathrm{O}(W_M(n))$. On the other hand

$$\begin{pmatrix} \mathrm{I} & A & 0 \\ 0 & \mathrm{I} & B \\ 0 & 0 & \mathrm{I} \end{pmatrix}^{-1} = \begin{pmatrix} \mathrm{I} & -A & AB \\ 0 & \mathrm{I} & -B \\ 0 & 0 & \mathrm{I} \end{pmatrix}$$

and thus $W_M(n) \in \mathrm{O}(W_I(n))$ for an optimal multiplication algorithm. □

Note, that inequality (2) is very loose. Taking into account that practically $W_M(n) >> n^2$ it is rather $W_S(n) \approx 2W_M(n)$.

2.1.5 Parallel Time

Figure 2 shows the DAG for Strassen's inversion algorithm. It can be easily seen, that all expensive operations are on the critical path. Similar to the operation count, I get a recurrence for the parallel time.

$$T_S(n) = 2T_S\left(\frac{n}{2}\right) + 4T_M\left(\frac{n}{2}\right)$$
$$T_S(1) = 1$$

For a time-optimal multiplication subroutine with $T_M(n) \in \mathrm{O}(\log n)$, Strassen's inversion algorithm is in $\mathrm{O}(n \log n)$ just as Gaussian elimination.

$$\begin{aligned} T_S(n) &= 2\left(T_S\left(\frac{n}{2}\right) + 2T_M\left(\frac{n}{2}\right)\right) \\ &= nT_S(1) + \sum_{i=1}^{\log n} 2^i \cdot 2T_M\left(\frac{n}{2^i}\right) \\ &\le nT_S(1) + \sum_{i=1}^{\log n} 2^i \cdot 2T_M(n) \\ &= nT_S(1) + 4T_M(n)\sum_{i=1}^{\log n} 2^{i-1} \\ &= nT_S(1) + 4T_M(n)\frac{1-2^{\log n}}{1-2} \\ &= nT_S(1) + 4T_M(n)(n-1) \\ &\in \mathrm{O}(n \log n) \end{aligned}$$

$$A = \begin{pmatrix} B & C^T \\ C & D \end{pmatrix} \quad A^{-1} = \begin{pmatrix} B^{-1} + B^{-1}C^TS^{-1}CB^{-1} & -B^{-1}C^TS^{-1} \\ -S^{-1}CB^{-1} & S^{-1} \end{pmatrix}$$

$$\text{with: } S = D - CB^{-1}C^T$$

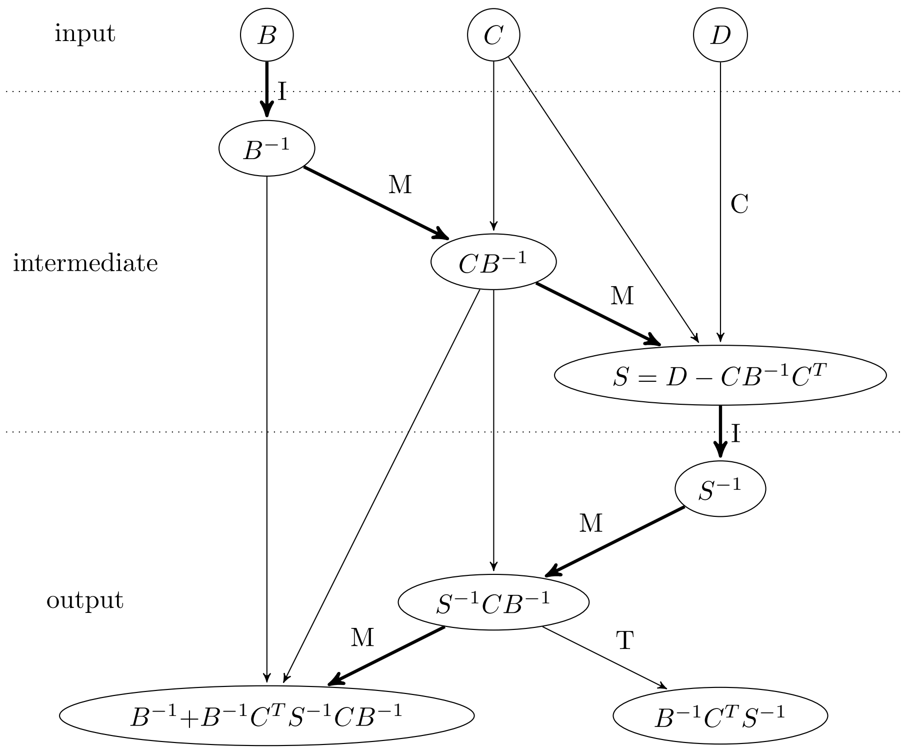

Figure 2: DAG of Strassen's matrix inversion algorithm.
The critical path ($\longrightarrow$) consists of both recursive inversions (I) and all four multiplications (M). The only operations outside of the critical path are one copy (C) and one transpose (T) which are relatively cheap. Further addition, transposition, and scaling operations can be done on-the-fly during the multiplications.

However, the critical path is still in $\Omega(n)$:

$$T_S(n) \geq 2T_S\left(\frac{n}{2}\right) = nT_S(1) \in \Omega(n)$$

Remark On mediocre parallel systems, the length of the critical path is not that important. They benefit from the main work happening in few, large matrix multiplications which parallelize way better than the row operations in Gaussian elimination.

2.1.6 Storage Requirements

In addition to the input and output matrix, Strassen's inversion algorithm requires storage for $C\tilde{A}$ and S plus one recursive inversion (compare the pseudocode in figure 1). We get the recurrence for the additional storage $S_S(n)$

$$\begin{aligned} S_S(n) &= 2\left(\frac{n}{2}\right)^2 + S_S\left(\frac{n}{2}\right) \\ S_S(1) &= 0 \end{aligned}$$

which solves to

$$\begin{aligned} S_S(n) &= 2\sum_{i=1}^{\log n-1}\left(\frac{n}{2^i}\right)^2 \\ &= \frac{1}{2}n^2\sum_{i=0}^{\log n-2}\frac{1}{4^i} \\ &\leq \frac{1}{2}n^2\frac{1}{1-\frac{1}{4}} \\ &= \frac{2}{3}n^2 \end{aligned}$$

Thus, Strassen's inversion algorithm needs overall storage for about $\frac{8}{3}n^2$ elements.

2.2 Matrix Inversion by Iterative Newton Approximation

Newton iterations can be used to approximate the inverse of a matrix. Given a suitable initial approximation, the error converges quadratically so that the "number of correct bits" doubles with each iteration. Because Newton approximation, except for the cheap initialization step, consists solely of full size multiplications, it parallelizes well. It does, however, require a non-constant number of iterations, each of which is as much work as complete inversion by an optimal algorithm. Therefore it is not work-optimal.

Because numerical errors are automatically corrected, Newton approximation is popular for its numerical stability. Said stability will be tested in the experimental part of this thesis, particularly in sections 5.3 and 5.4.

2.2.1 Algorithm

The Newton approximation algorithm makes use of the following error measure. Let X be an approximation of M^{-1}.

Let
$$\begin{aligned} \mathrm{R}(X) &= \mathrm{I} - XM \\ \mathrm{err}(X) &= \|R(X)\| \end{aligned}$$

For an approximation X_i the next improvement is given by

$$\begin{aligned} X_{i+1} &= X_i + \mathrm{R}(X_i) \cdot X_i \\ &= (2\mathrm{I} - X_i M) X_i \ . \end{aligned} \tag{3}$$

This at least squares the error:

$$\begin{aligned} \mathrm{err}(X_{i+1}) &= \|\mathrm{I} - (2\mathrm{I} - X_i M) X_i M\| \\ &= \|\mathrm{I}^2 - 2X_i M + (X_i M)^2\| \\ &= \|(\mathrm{I} - X_i M)^2\| \\ &\le \|\,\mathrm{R}(X_i)\|^2 \\ &= \mathrm{err}(X_i)^2 \end{aligned} \tag{4}$$

It is not obvious how to efficiently calculate an initial approximation X_0 with $\mathrm{err}(X_0) < 1$. I make use of the method described by Pan and Reif.[10] For a general non-singular matrix $\bar{M}$, choose

$$\bar{X}_0 = \bar{t}\mathrm{I} \quad \text{with } \bar{t} = \frac{1}{\|\bar{M}\|_1 \cdot \|\bar{M}\|_\infty} = 1 \Big/ \max_i \sum_j |\bar{m}_{ij}| \cdot \max_j \sum_i |\bar{m}_{ij}| \ .$$

In case of a symmetric positive definite matrix M it suffices to choose

$$X_0 = t\mathrm{I} \quad \text{with } t = \frac{1}{\|M\|_\infty} = 1 \Big/ \max_i \sum_j |m_{ij}| \ . \tag{5}$$

Theorem 2.8.
X_0 as defined by equation (5) for a symmetric positive definite matrix M satisfies

$$\mathrm{err}(X_0) \le 1 - \frac{1}{\sqrt{n}\,\mathrm{cond}\, M} \ .$$

The proof makes use of the following lemma.

Lemma 2.9.
Let X_0 and t be as defined in equation (5). Then μ is eigenvalue of $\mathrm{R}(X_0) = \mathrm{I} - X_0 M$ exactly if $\frac{1-\mu}{t}$ is eigenvalue of M.

Proof.
Let $\mathbf{v}$ be an eigenvector to μ.

$$\begin{aligned} & & \mu\mathbf{v} &= \mathrm{R}(X_0)\mathbf{v} \\ &\Leftrightarrow & \mu\mathbf{v} &= \mathbf{v} - X_0 M\mathbf{v} \\ &\Leftrightarrow & \mu\mathbf{v} &= \mathbf{v} - tM\mathbf{v} \\ &\Leftrightarrow & \frac{1-\mu}{t}\mathbf{v} &= M\mathbf{v} \end{aligned}$$

□

Proof of theorem 2.8.
I make use of the fact, that for a diagonalizable matrix its norm is its largest eigenvalue. Since M is diagonalizable, so is $\mathrm{R}(X_0)$. Let μ be the largest eigenvalue of $\mathrm{R}(X_0)$. Then by lemma 2.9 $\frac{1-\mu}{t}$ is the smallest eigenvalue of M and by lemma 2.4 $\frac{t}{1-\mu} = \|M^{-1}\|$.

$$\begin{aligned} && \frac{1-\mu}{t} &= \frac{1}{\|M^{-1}\|} \\ \Rightarrow && 1-\mu &= \frac{t}{\|M^{-1}\|} \\ \Rightarrow && \operatorname{err}(X_0) = \|\mathrm{R}(X_0)\| = \mu &= 1 - \frac{t}{\|M^{-1}\|} \end{aligned}$$

On the other hand, it is

$$\frac{\|M^{-1}\|}{t} = \|M^{-1}\| \cdot \|M\|_\infty \leq \|M^{-1}\| \cdot \sqrt{n}\|M\| = \sqrt{n} \operatorname{cond} M \ .$$

And thus

$$\operatorname{err}(X_0) \leq 1 - \frac{1}{\sqrt{n} \operatorname{cond} M} \ .$$

□

It remains the question, how many iterations are necessary to reach a desired bound on the error ϵ.

Theorem 2.10.
Starting from X_0 as defined by equation (5) for a symmetric positive definite matrix M, k iterations of Newton approximation suffice to ensure $\operatorname{err}(X_k) \leq \epsilon$, *with*

$$k = \frac{1}{2} \log n + \log \log \frac{1}{\epsilon} + \log \operatorname{cond} M \ .$$

Lemma 2.11.
For $a > 1$ hold:

$$\begin{aligned} \frac{1}{a} \leq \log_e \frac{a}{a-1} &\leq \frac{1}{a-1} \\ \frac{1}{a} \leq \frac{1}{a \log_e 2} \leq \log_2 \frac{a}{a-1} &\leq \frac{1}{(a-1)\log_e 2} \end{aligned}$$

Proof.

$$\begin{aligned} & \forall x \in [a-1, a] : \frac{1}{a} \leq (\log_e x)' \leq \frac{1}{a-1} \\ \Rightarrow \quad & \frac{1}{a} \leq \log_e(a) - \log_e(a-1) \leq \frac{1}{a-1} \end{aligned}$$

□

Proof of theorem 2.10.

Let $\operatorname{err}(X_0) = \xi$

By theorem 2.8 and lemma 2.11 it is

$$\log\frac{1}{\xi} \geq \log\frac{\sqrt{n}\operatorname{cond} M}{\sqrt{n}\operatorname{cond} M - 1} \geq \frac{1}{\sqrt{n}\operatorname{cond} M} \tag{6}$$

$$\begin{aligned}
\text{Then} \qquad & k = \log\log\frac{1}{\epsilon} + \frac{1}{2}\log n + \log\operatorname{cond} M \\
& \quad = \log\log\frac{1}{\epsilon} + \log(\sqrt{n}\operatorname{cond} M) \\
& \quad = \log\log\frac{1}{\epsilon} - \log\frac{1}{\sqrt{n}\operatorname{cond} M} \\
\overset{(6)}{\Rightarrow} \qquad & k \geq \log\log\frac{1}{\epsilon} - \log\log\frac{1}{\xi} \\
& \quad = \log\frac{\log\epsilon}{\log\xi} \\
\Leftrightarrow \qquad & 2^k \geq \frac{\log\epsilon}{\log\xi} \\
\Leftrightarrow \qquad & \underbrace{\log\xi}_{<0} \cdot 2^k \leq \log\epsilon \\
\Leftrightarrow \qquad & 2^{\log\xi\cdot 2^k} \leq \epsilon \\
\Leftrightarrow \qquad & \xi^{2^k} \leq \epsilon \\
\overset{(4)}{\Rightarrow} \qquad & \operatorname{err}(X_k) \leq \epsilon
\end{aligned}$$

□

To simplify further calculations, in the following I estimate $k \leq \alpha \log n$. Usually, one would want the output as precise as possible with the used number format, what makes ϵ a constant of the machine. If a certain output precision is necessary for further calculations, then it will depend on n and $\operatorname{cond} M$. As long as $\frac{1}{\epsilon}$ and $\operatorname{cond} M$ are polynomial in n, $\alpha \log n$ is an upper bound for k. Furthermore, other, non approximative algorithms do not depend on the condition of M, so it is unclear how to compare with them without assumptions about the condition.

2.2.2 Work

Computing the initial inverse takes n^2 operations. One iteration of Newton approximation requires two matrix multiplications and one addition. By theorem 2.10, $k \leq \alpha \log n$ iterations suffice. Therefore the number of operations necessary, $W_N(n)$, is:

$$\begin{aligned}
W_N(n) &\leq (2W_M(n) + n^2)\cdot\alpha\log n + n^2 \\
&\in \mathrm{O}(W_M(n)\cdot\log n)
\end{aligned}$$

By the factor of $\log n$ Newton approximation is not work-optimal.

2.2.3 Parallel Time

All high level steps of Newton inversion have to be done in serial. Computation of the initial approximation is binary reduction of n^2 elements and can be done in $2\log n$ timesteps. Addition is completely parallelizable and needs only one timestep, that I will neglect. Thus

$$\begin{aligned} T_N(n) &\leq 2T_M(n) \cdot \alpha \log n + 2\log n \\ &\in \mathrm{O}(T_M(n) \cdot \log n) \end{aligned}$$

With a time-optimal multiplication subroutine with $T_M(n) \in \mathrm{O}(\log n)$, Newton approximation is a polylog time algorithm.

2.2.4 Storage Requirements

In addition to the input and output matrix, Newton approximation requires storage for the product $X_i M$ and the improved approximation X_{i+1}. With a multiplication subroutine that needs no additional storage, it requires storage for $4n^2$ elements total. However, a polylog multiplication needs a constant amount of storage per processor which is in $\Omega(\frac{W_M(n)}{\log n})$ for logarithmic runtime.

3 A Work-Optimal Polylog Time Algorithm

In the previous section I have shown how Strassen's inversion algorithm uses matrix multiplications to effectively parallelize most of the work without increasing the amount over a constant factor. Still, it has a critical path length of n, because it can not effectively break up the small inversions that origin from its recursive nature. Newton approximation, on the other hand, consists only of sub-steps that operate on the full input size and can be parallelized effectively (namely multiplications, additions, and reductions). Unfortunately, it requires a $\log n$ factor more work.

A work-optimal, yet polylog parallelizable algorithm OPT can be created by combining the two algorithms in the following way: Use Strassen's algorithm for the large size inversions, but use Newton approximation once the recursion comes to small sizes. That way Strassen's algorithm works on the parts it is good at parallelizing, and Newton approximation gets small inputs and thereby requires small amounts of additional work.

3.1 Algorithm

For the remainder of this section let

$$n' = \frac{n}{\log n} \ .$$

Obtain OPT by modifying Strassen's algorithm for symmetric positive definite matrices to a base size of $\leq n'$ instead of 1. For the base inversions use Newton approximation.

From theorem 2.2 we know that all base inversions are symmetric positive definite. Furthermore, from theorems 2.3 and 2.6 we know that no condition will exceed that of the input matrix M.

3.2 Work

OPT gets the recurrence for the operation count from Strassen's algorithm but with Newton approximation as base case. The operation count is again for a symmetric positive definite input matrix M.

$$\begin{aligned}
W_O(n) &= 2W_O\left(\frac{n}{2}\right) + 4W_M\left(\frac{n}{2}\right) + 2\left(\frac{n}{2}\right)^2 \\
W_O(n') &= W_N(n') \\
&\leq (2W_M(n') + n'^2) \cdot \alpha \log n' + n'^2 \\
&\leq (2W_M(n') + n'^2) \cdot \alpha \log n \qquad \text{for } n^\alpha \geq 4
\end{aligned}$$

The recurrence solves to:

$$\begin{aligned}
W_O(n) &= 2\left(W_O\left(\frac{n}{2}\right) + 2W_M\left(\frac{n}{2}\right) + \left(\frac{n}{2}\right)^2\right) \\
&= \log n \cdot W_O(n') + \underbrace{\sum_{i=1}^{\log\log n} 2^i \left(2W_M\left(\frac{n}{2^i}\right) + \left(\frac{n}{2^i}\right)^2\right)}_{=:f}
\end{aligned}$$

$$\begin{aligned}
f &\leq \sum_{i=1}^{\log\log n} 2^i \left(\frac{2}{4^i} W_M(n) + \frac{n^2}{4^i}\right) \\
&= \sum_{i=1}^{\log\log n} \frac{2}{2^i} W_M(n) + \frac{n^2}{2^i} \\
&= \left(W_M(n) + \frac{n^2}{2}\right) \sum_{i=1}^{\log\log n} \left(\frac{1}{2}\right)^{i-1} \\
&= \left(W_M(n) + \frac{n^2}{2}\right) \frac{1 - \left(\frac{1}{2}\right)^{\log\log n}}{1 - \frac{1}{2}} \\
&\leq \left(W_M(n) + \frac{n^2}{2}\right) \frac{1}{1 - \frac{1}{2}} \\
&= 2W_M(n) + n^2
\end{aligned}$$

$$\begin{aligned}
\log n \cdot W_O(n') &\leq \alpha \log^2 n \cdot \left(2W_M\left(\frac{n}{\log n}\right) + n'^2\right) \\
&= \alpha \log^2 n \cdot \left(\frac{1}{\log^2 n} 2W_M(n) + \frac{n^2}{\log^2 n}\right) \\
&= 2\alpha W_M(n) + \alpha n^2
\end{aligned}$$

$$\begin{aligned}
\Rightarrow \qquad W_O(n) &\leq 4\alpha W_M(n) + 2\alpha n^2 \qquad \text{for } \alpha \geq 1 \\
&\in \mathrm{O}(W_M(n)) = \mathrm{O}(W_I(n))
\end{aligned}$$

Thus, this algorithm is work-optimal.

Remark If a non-symmetric matrix is inverted with the technique described in section 2.1.3, two more multiplications are necessary. Still, OPT is work-optimal with that extension.

3.3 Parallel Time

As with the operation count, the recurrence for the parallel time comes from Strassen's algorithm with the base case from Newton approximation.

$$\begin{aligned}
T_O(n) &= 2T_O\left(\frac{n}{2}\right) + 4T_M\left(\frac{n}{2}\right) \\
T_O(n') &= 2T_M(n') \cdot \alpha \log n' + 2\log n' \\
&\leq 2T_M(n) \cdot \alpha \log n && \text{for } n^\alpha \geq 4
\end{aligned}$$

The recurrence solves to:

$$\begin{aligned}
T_O(n) &= 2\left(T_O\left(\frac{n}{2}\right) + 2T_M\left(\frac{n}{2}\right)\right) \\
&= \log n \cdot T_O(n') + \underbrace{\sum_{i=1}^{\log\log n} 2^i \cdot 2T_M\left(\frac{n}{2^i}\right)}_{=:f}
\end{aligned}$$

$$\begin{aligned}
f &\leq \sum_{i=1}^{\log\log n} 2^i \cdot 2T_M(n) \\
&= 4T_M(n) \sum_{i=1}^{\log\log n} 2^{i-1} \\
&= 4T_M(n) \frac{1 - 2^{\log\log n}}{1-2} \\
&= 4T_M(n)(\log n - 1)
\end{aligned}$$

$$\begin{aligned}
\Rightarrow \quad T_O(n) &\leq 6T_M(n) \cdot \alpha \log^2 n \\
&\in \mathrm{O}(T_M(n) \cdot \log^2 n)
\end{aligned}$$

With a time-optimal multiplication subroutine with $T_M(n) \in \mathrm{O}(\log n)$, this is a polylog time algorithm.

Remark Just as with the work-optimality, inverting a non-symmetric matrix with the technique from section 2.1.3 does not break the polylog time parallelizability of OPT.

3.4 Numerical Stability

Strassen's inversion algorithm is unpopular for numerical instability. Baley and Ferguson experimented with using iterations of Newton approximation to improve the result.[11] They tried three variants of extra effort to put into the improvement: First, they applied one iteration of Newton approximation on the final output of Strassen's algorithm. Second, they applied one iteration on the output of every level of Strassen's algorithm except on final result. Third, they combined both.

This approach does neither break the property of work-optimality nor of polylog parallelizability (of OPT), since it only adds a constant number of matrix multiplications where there already are some.

```
input: $M = \begin{pmatrix} A & C^T \\ C & B \end{pmatrix}$
$\tilde{A}$ = $\mathrm{Inv}(A)$
$C\tilde{A}$ = $C \cdot \tilde{A}$
$S$ = $B - C\tilde{A} \cdot C^T$
$\tilde{S}$ = $\mathrm{Inv}(S)$
$P$ = $-\tilde{S} \cdot C\tilde{A}$
$R$ = $\tilde{A} - (C\tilde{A})^T \cdot P$
$M'$ = $\begin{pmatrix} R & P^T \\ P & \tilde{S} \end{pmatrix}$
$\tilde{M}$ = $(2I - M'M)M'$
output: $\tilde{M}$
```

Figure 3: Pseudocode for OPT-S.

The results of their experiments where promising. While the output of Strassen's algorithm was quite erroneous, each variant of effort improved the result significantly. The third variant even produced better results than the implementation of Gaussian elimination they used for reference. It remains to say, that in addition to Strassen's inversion algorithm they also used Strassen's matrix multiplication algorithm.

I will proof that the second variant, which I call OPT-S, is sufficient to correct errors that origin in the inversion. To do that I assume that multiplications in one level of Strassen's algorithm do only magnify the errors of recursive inversions, and errors of the multiplications themselves can be neglected. Furthermore, I do a first order error analysis, i.e. I neglect any squares of errors.

The pseudocode for OPT-S (see figure 3) results by adding the iteration of Newton approximation to the pseudocode for Strassen's algorithm (compare figure 1). For consistency in the error calculation, I renamed R to $\tilde{A}$ for the period where it holds the inverse of A. Note, that for the second variant the modified code is not to be applied for the outermost recursion.

Since I want a bound for the error of an inversion, I assume the recursive inversion to satisfy error bounds χ_X.

For $X \in \{A, S, M\}$ let $\chi_X = \|\tilde{X} - X^{-1}\|$.
Let $\bar{f}$ be the exact (arithmetic) value of f.
Let $\delta(f) = \|f - \bar{f}\|$.

Note: χ_S does not include the error $\delta(S)$ from the calculation of S but only the error from the recursive inversion.

Theorem 3.1.

Let $$\epsilon = \frac{1}{\mathrm{cond}^{10} A}$$
Then $\chi_A, \chi_S \leq \epsilon$ *suffices for* $\chi_M \leq \epsilon$

Proof.

Let $$\Lambda = \|M\|$$

$$\begin{aligned}
\lambda &= \|M^{-1}\| \\
\kappa &= \operatorname{cond} M = \Lambda\lambda
\end{aligned}$$

Then

$$\begin{aligned}
\delta(\tilde{A}) &= \chi_A \le \epsilon \\
\delta(C\tilde{A}) &\le \|C\|\delta(\tilde{A}) \le \epsilon\Lambda \\
\delta(S) &\le \delta(C\tilde{A})\|C^T\| \le \epsilon\Lambda^2 \\
\delta(\tilde{S}) &= \|\tilde{S} - \bar{S}^{-1}\| \\
&\le \|\tilde{S} - S^{-1}\| + \|S^{-1} - \bar{S}^{-1}\| \\
&\le \chi_S + \|S^{-1}\|\|\bar{S} - S\|\|\bar{S}^{-1}\| \\
&\le \epsilon + \delta(S)\lambda^2 \\
&\le \epsilon + \epsilon\kappa^2 \\
\delta(P) &\le \delta(\tilde{S})\|C\tilde{A}\| + \|\tilde{S}\|\delta(C\tilde{A}) \le 2\epsilon\kappa + \epsilon\kappa^3 \\
\delta(R) &\le \delta(\tilde{A}) + \delta(C\tilde{A})\|P\| + \|C\tilde{A}\|\delta(P) \\
&\le \epsilon + \epsilon\kappa + 2\epsilon\kappa^2 + \epsilon\kappa^4 \\
\delta(M') &\le \delta(R) + 2\delta(P) + \delta(\tilde{S}) \\
&\le 2\epsilon + 3\epsilon\kappa + 3\epsilon\kappa^2 + \epsilon\kappa^3 + \epsilon\kappa^4 \\
&\le 2\epsilon\kappa^4 && \text{for } \kappa \ge 3
\end{aligned}$$

From equation (4) we know

$$\begin{aligned}
&& \operatorname{err}(\tilde{M}) &\le \operatorname{err}(M')^2 \\
\Leftrightarrow && \|\mathrm{I} - \tilde{M}M\| &\le \|\mathrm{I} - M'M\|^2 \\
\Rightarrow && \|M^{-1} - \tilde{M}\|\frac{1}{\|M^{-1}\|} &\le \|M^{-1} - M'\|^2\|M\|^2 && (7) \\
\Leftrightarrow && \|M^{-1} - \tilde{M}\| &\le \|M^{-1} - M'\|^2\|M\|^2\|M^{-1}\| \\
&& &= \|M^{-1} - M'\|^2\Lambda^2\lambda
\end{aligned}$$

Thus

$$\begin{aligned}
\chi_M = \delta(\tilde{M}) &\le \delta(M')^2\Lambda^2\lambda \\
&\le 4\epsilon^2\Lambda\kappa^9 \\
&\le \frac{4\Lambda\kappa^9}{\kappa^{20}} && \text{for } \lambda \ge 4 \\
&\le \epsilon
\end{aligned}$$

□

Remark Inequation (7) is very loose. It is not possible to proof any better bound in that step using only the axioms of matrix norms. Still, a much lower error bound ϵ should be necessary in practice.

3.5 Storage Requirements

Just as for Newton approximation, the storage requirements depend on the multiplication subroutine. With no additional storage for the multiplication, OPT needs about the same amount of storage as Strassen's inversion algorithm, but logarithmic runtime requires an amount of processors and storage in $\Omega(\frac{W_M(n')}{\log n'})$.

4 Implementation

Purpose of my implementation was to do the experiments described in the next section. It is structured as a full linear algebra library but only contains the operations required for the experiments. For example, there is no `operator +` defined, but an operation `add` that computes $A = \alpha A + \beta B$. Additionally, there are special functions for operations that would be less efficient when combined of standard operations, e.g. `subtract_from_alpha_unity` that computes $A = \alpha \mathrm{I} - A$.

Mainly, the implementation consists of a `matrix` container class templated to the type of an element, and a test program that parses the command line, generates pseudorandom input and runs the tests. The `matrix` class offers high level operations, as well as low level access for operations that require it for efficiency. Built on the `matrix` class is an, again templated, class `matrix_inverse`. Its only data member is a `matrix` to hold the inverse, but its constructor takes the `matrix` to invert and some parameters and performs the inversion. An instance of `matrix_inverse` can be used as `matrix` via its `operator matrix`.

To generate input for the experiments, a set of `static` methods is grouped in the templated `matrix_generator` class. It includes pseudorandom generation with the desired condition as parameter and some matrix families to test correctness of the matrix operations. As pseudorandom number generator and for the distribution I use `boost::random` from the Boost C++ librarys.[20]

4.1 BLAS and LAPACK Interface

Strassen's inversion algorithm and inversion by Newton approximation build on matrix multiplication. For an efficient multiplication subroutine, I decided to link to a BLAS library.

BLAS is short for "Basic Linear Algebra Subprograms" and refers to an almost standard interface for a set of vector and matrix operations. Unfortunately, even though BLAS is centrally defined by Netlib,[16] there is neither a central, complete declaration of the interface, nor documentation of its exact functionality, nor unique naming of the functions. Some libraries use function names in capitals, others prefix the names with an underscore. Some even have two versions of the same function, one which interprets matrices as stored in column-major, the other in row-major. Due to the inconsistent naming issues, the program code can not simply be linked to different libraries, but needs to be adapted. Adapting is further complicated by the inconsistent functionality and incomplete documentation. For example, ATLAS's[18] documentation simply refers to Netlib's documentation of BLAS, which to begin with has incomplete descriptions of the functions effects and secondly defines other function names than implemented by ATLAS. The commercial product Intel MKL alone is much more comfortable to use. It includes a manual[19] that describes the effects and parameters of every function in detail. The only exception is, which dimension parameter refers to which dimension of the matrices, what I managed to only find out by interpreting the descriptions of different error cases.

Generally, every function has four names in BLAS for the four floating point element types single and double precision, real and complex values, denoted by the first part of the name being `s`, `d`, `c`, or `z`. The last part of the functions names state the actual operation, while the middle part, depending on the

0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

general matrix
column-major storage
(`ge`)

0	-	-	-
1	5	-	-
2	6	10	-
3	7	11	15

entrys above the main
diagonal ignored
(`sy`, lower)

0	-	-	-
1	4	-	-
2	5	7	-
3	6	8	9

packed storage of the
lower triangle
(`sp`, lower)

Figure 4: Overview over the three storage schemes for symmetric matrices defined by BLAS. Numbers are the positions in the array at which the matrix elements are stored.

`dgemm`	`dsymm`	`dsyrk`
$C = \alpha A \cdot B + \beta C$	$C = \alpha S \cdot A + \beta C$	$S = \alpha A \cdot A^T + \beta S$
$C = \alpha A^T \cdot B + \beta C$	$C = \alpha A \cdot S + \beta C$	$S = \alpha A^T \cdot A + \beta S$
$C = \alpha A \cdot B^T + \beta C$		
$C = \alpha A^T \cdot B^T + \beta C$		

with A, B, and C general matrices, S a symmetric matrix,
and α and β scalars

Figure 5: Overview over the matrix multiplication operations for general and symmetric matrices available in BLAS.

operation, distinguishes between special properties the input matrices may have. Those properties include symmetry or symmetric positive definiteness, including special storage formats for symmetric rsp. hermitian matrices. Because BLAS libraries are usually implemented in Fortran, general matrices are assumed to be stored in column-major in a single array as is default in Fortran. Symmetric matrices can be stored either just as general matrices, or in the same format as general matrices but with entries above or below the main diagonal ignored, or in packed storage that skips the entries above/below the main diagonal, denoted by `ge`, `sy`, and `sp` respectively (compare figure 4).

Matrix multiplication routines are available in BLAS for general input matrices, which can be transposed on-the-fly, and for one symmetric and one general matrix but without the possibility to transpose. Both routines add the product to the previous content of the output matrix and apply scalar factors to both summands (compare figure 5). Using the routine for one symmetric input matrix mainly has the advantage, that only one triangle of said matrix has to be written and does not need to be transposed to set the other entries. Unfortunately, there is no routine that computes a matrix product that is known to be symmetric beforehand. The only routine that explicitly outputs a symmetric matrix, `dsyrk`, can only multiply a matrix with its own transpose.

LAPACK is an extension to the BLAS standard and a reference implementation build on it that adds routines for solving systems of linear equations, least squares problems, eigenvalue and singular value problems, and related tasks. It includes matrix inversion via LU factorization rsp. Cholesky factorization, which I use as reference for runtime and numerical stability.

My implementation contains a wrapper layer for BLAS and LAPACK. Its main function is to bridge between the templated `matrix` class and the typespecific BLAS and LAPACK names via template specialization. Additionally it adapts from row-major matrix storage to the form required by the library and takes care of ordering of the dimension arguments. Adapting between row-major and column-major is efficient and only requires not to get confused by the ordering and naming of parameters. For example, to calculate $C = A \cdot B$, one can instead calculate $C^T = B^T \cdot A^T$. The transposition happens implicitly by the reinterpretation from row-major to column-major. Only proper swapping of the matrix and dimension parameters is required. So, the second function of the BLAS wrapper layer boils down to avoiding confusion.

4.2 Internal Data Structures

The `matrix` class internally uses a low level storage scheme that I designed to easily allow use of BLAS subroutines as well as efficient algorithms for the functions not included in BLAS. First of all, a matrix is represented by integers for the `height` and `width` and an array to hold its elements. The array is managed by a `boost::scoped_array` from the Boost C++ libraries.[20] The elements are stored in row-major. Other orders like Morton order may be more efficient to use with recursive algorithms, but are not supported by BLAS.[8] In addition to the size and element storage, the `matrix` class contains integers for the `offset` of the first matrix element in the array and for the `row-length` in the array (i.e. the distance in the array of an element and the one below in the matrix) as well as a flag indicating that the matrix is known to be `symmetric`.

The `offset` and `row-length` members allow efficient construction of submatrices. A constructor for this purpose copies the `scoped_array` (i.e. pointer and reference counter, not the actual elements) and the `row-length` and and sets `offset` and `symmetric` appropriately. That way, no extra storage is required for the elements of the submatrix and changes to them automatically affect the supermatrix as well. The `scoped_array` takes care of freeing the array once the supermatrix and all its (direct and indirect) submatrices have been deleted. The technique of height, width, start (= array-pointer + offset), and row-length is directly supported by BLAS.

Managing the array of elements with a `boost::scoped_array` also allows to easily implement an efficient copy constructor and assignment operator. Both copy the members of the `matrix` what again means to only copy the pointer and reference counter for the element array but not the actual elements. If the original `matrix` object is destroyed afterwards, the `boost::scoped_array` simply decreases the reference counter, thus effectively transmitting ownership of the array from the original to the new object.

4.3 Multiplication

The `multiply` member function of the `matrix` class offers the full functionality of both BLAS routines, `dgemm` for two general and `dsymm` for one symmetric and one general input matrix.[9] If an input matrix has its `symmetric` flag set, it is ignored whether it shall be transposed. If none of the input matrices shall

[8]The only other order supported by BLAS apart from row-major is column-major.

[9]dgemm and dsymm are for double precision matrix multiplication. For single precision or complex numbers my BLAS wrapper layer uses sgemm and ssymm, cgemm and csymm, or zgemm and zsymm as appropriate.

be transposed and at least one of them has its `symmetric` flag set, `dsymm` is used with the first symmetric matrix as the symmetric input. That way, this matrix's entries above the main diagonal are ignored and need not be set by the step producing the matrix. In all other cases, `dgemm` is called.

I defined a second multiplication member function `multiply_symmetric` for the case that the result it known to be symmetric beforehand. I will call a multiplication with this additional information "symmetric multiplication", although it does not imply commutativity or any symmetry in the factors. In its final implementation, it does nothing more than call `multiply` and set the `symmetric` flag.

Examination of the formulas that lead to OPT shows, that exactly every other multiplication is symmetric. Obviously, making use of this fact can save almost one fourth of the calculation (the diagonals still have to be computed completely). Unfortunately, BLAS does not support use of this knowledge. I tried implementing a recursive matrix multiplication routine that can take advantage of the symmetry of a multiplication. For a comparable efficient implementation it used BLAS multiplication as base case. Unfortunately there seems to be a bug in the Intel MKL library that made the implementation unusable.[10] When multiplications into different submatrices of the same result-matrix where started by multiple parallel threads, the output had significantly large errors. This did not happen, when multiple threads where started but all ran on a single core.

4.4 Other Parallel Matrix Operations

All matrix operations that operate on the elements are implemented to be able to use parallelism. Those that just do a single pass over the elements, like addition and initialization, use `pragma parallel for`.

Reductions like one- and infinity-norms and the maximum absolute element that is used as error measure by Newton approximations require special attention. To find the maximum in single threaded execution, one just stores the maximum element so far in a variable. Doing so with multiple threads would cause bugs due to unsynchronized writing and heavy false sharing. Instead, every thread needs its own variable for the maximum element it found so far. That is simple to implement by defining the variable inside a `pragma omp parallel` block, what also puts it on the treads own stack. Afterwards, each thread updates a global maximum variable in a synchronized block.

Operations that work recursively, like transposition and the failed symmetric multiplication, pass an additional parameter down the recursion on how many threads they should parallelize. Each time the recursion splits the task into independent subtasks of equal size, the number of threads is divided accordingly. The subtasks are parallelized with `pragma parallel sections`. Once the number of threads to split on reaches one, the recursion switches to a purely serial instance of the function. This prevents recursive calls in low levels, of which there are very many, to spend time in unnecessary calls to OpenMP. Parallelizing is also stopped, if the size falls below a limit given by a preprocessor macro. The value was automatically adjusted, see section 4.7. The number of

[10]The bug also occurred when linking to the GNU OpenMP library instead of the Intel OpenMP library.

threads is set to the number of available processors by the outermost call of the function.

The `matrix` class provides a member function `partition` that partitions the matrix into four submatrices and returns them in a structure. The provided partitioning scheme is used by all recursive algorithms including Strassen's inversion algorithm and OPT.

Transposition and the similar `make_symmetric`, that fills the upper triangle with the transpose of the lower triangle, work recursively with the scheme provided by `partition`. For very small matrices they switch to simple nested `for`-loops, that are more efficient in this case.[6] The size is again automatically adjusted.

4.5 Inversion

The `matrix_inverse` class capsules all inversion methods. Its constructor takes two arguments: The matrix to invert and a string that specifies the combination of inversion methods to use.

4.5.1 Recursion Management

To allow higher flexibility for experiments, the recursive inversion functions do neither take a parameter for the recursion depth or have it fixed, nor do they use a fixed base inversion. Instead, a `recursion` object holds a list of function pointers to the inversion functions to use. Recursive inversion functions perform only one level of recursion and then call the `recursion` object via a callback function object. The `recursion` object then calls them again or a different function as desired. So that all inversion functions are of the same type, other inversion functions are capsuled in wrappers that also take the callback object but ignore it.

The function pointer list is initialized by parsing the string passed to the `matrix_inverse` constructor. Each entry corresponds to one level in the tree of recursive calls with non-recursive methods as leaves. Because all recursive calls are run in serial, the tree is traversed in pre-order. Thus a single iterator suffices to find the inversion function to use; it is increased after starting the call and decreased when the call returns. Thereby its position in the list always corresponds to the current level in the recursion tree.

The ability to specify the method used for every level of recursion separately allows not only to compose Strassen's inversion algorithm, OPT, and OPT-S, but also to make alterations at runtime. For example, to get OPT one would specify the appropriate number of levels of Strassen's inversion algorithm and then Newton approximation. If one wishes, the number of levels can simply be changed. As another example, OPT-S results by inserting single iterations of Newton approximation in between the levels of Strassen's inversion algorithm. This can be altered to do more than one iteration at a time or by inserting them only every k levels. Also, the LAPACK inversion (effectively MKL inversion) can be used as base case to OPT, to examine the effects of Strassen's algorithm without those of Newton approximation.

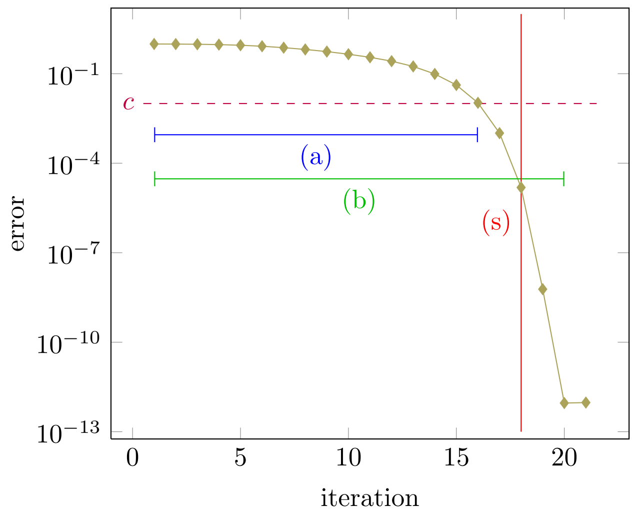

Figure 6: Typical evolution of the error value during a Newton approximation. Also shown are the maximal error constant c and the ranges in that condition (a) and (b) hold (compare section 4.5.2), as well as the point up to that single precision computation would be sufficient (s) (compare section 4.8).

4.5.2 Inversion Methods

The implementation of one level of Strassen's inversion algorithm follows the pseudocode from figure 1. Multiplications that result in a symmetric matrix use `multiply_symmetric`. Possible parallelism between the copy and other operations is not exploited. The transpose to compute P' is skipped completely and P' left undefined. As intermediate result, P' is never read due to the use of `dsymm` for further multiplications as described in section 4.3. The final result undergoes a call to `make_symmetric`, that fills in all missing parts. It would be cleaner to specify if P' is needed as a parameter to the function, but that would make the function itself and the callback object more complicated. Although `make_symmetric` sets some elements that where already correct from the base case of the inversion, the amount of additional work is neglectable compared to the rest of the inversion.

The implementation of Newton approximation uses, apart from the input matrix, one matrix `E` for the error matrix plus I and two `X`, `X_new` for the current and the new approximation. I follows the formula $X_{i+1} = (2\mathrm{I} - X_i M)X_i$. First, `E` is set to 2I, then $X_i M$ is multiplied and subtracted in one operation. Instead, one could subtract the product from 2I and store the difference in the same matrix. I expect both methods to be equally efficient, because they both involve a single pass over the elements with a conditional addition. In a few test runs, I found no difference. Both methods use a special member function of the `matrix` class that parallelizes as described in section 4.4. After the new approximation has been calculated, the contents of `Y` and `Y_new` are swapped (again including the array pointers but not the elements themselves).

The implementation of Newton approximation does not calculate the number of iterations by the formula of theorem 2.10. This spares the dependence on the condition as an input parameter and saves superfluous iterations. Instead, it measures the remaining error of each approximation as described by equation (8). To calculate the error measure, I is subtracted from `E` on-the-fly. During the tests, the maximum absolute element has shown to give good results. Anyway, the value calculated as error measure is not incorporated in the iterations in any way, but serves only to detect stagnation of improvement as termination condition. The iteration continues as long as at least one of two conditions holds: (a) The error of the new approximation is smaller than the previous to a certain power p. (b) The remaining error is larger than a fixed constant c. These conditions have shown to reliably detect two situations (compare figure 6): (a) When the border of representability is close, the error is no longer squared. Although little improvement is usually still possible, the iteration should stop because the effort is not worth it. p should be less than 2 to avoid misdetection due to numerical instability (e.g. 1.2). (b) At the beginning of the inversion the improvement is small, so that (b) is sometimes not met due to numerical instability. In that case the iterations shall still continue. c can be chosen rather large (e.g. 0.01).

The LAPACK standard defines the interface to the inversion of symmetric positive definite matrices to consist of two calls, `dpotrf` and `dpotri`. The first calculates a Cholesky factorization of the matrix. The second calculates the inverse from the Cholesky factorization. Both together are capsuled as one inversion method available in the `matrix_inverse` class. Those routines are specialized for symmetric positive definite matrices and can thus make use of all their benefits. I also implemented a function to call the versions for general matrices, `dgetrf` and `dgetri`, that work with an LU factorization, but do not use it for the experiments.

For the stabilization step of OPT-S, one iteration of Newton approximation is also available as pseudo-recursive function. It first uses the callback with the input (i.e. with the same size) to get an approximation and then performs the Newton iteration. To compose pure Strassen's inversion algorithm, simple inversion of 1×1 matrices is available as a separate function.

4.5.3 Singular Matrix Handling

The LAPACK inversion call `dgetri`, Newton approximation, and 1×1 inversion can detect singular input matrices. `dgetri` returns an error code, Newton approximation iterates infinitely, and 1×1 inversion divides by 0. In all cases (for Newton approximation a large number of iterations depending on n), an exception is thrown containing appropriate information. The exception is caught by the test program that records the failure in the output.

4.6 Profiling

For profiling, a global object of type `matrix_statistic` is available. The matrix operations, especially the inversion algorithms, call functions to store profiling data and to start and stop timing. By replacing some of those functions with empty ones, all overhead can be removed from the matrix operations trough compiler optimization. After an operation finished, the test program can read the profiling data from the `matrix_statistic` object. All profiling data of one test run is also available as a single line of text for printing to standard output. With these lines, the test data can easily be extracted and processed by evaluation scripts. This is especially useful for automated calculation of medians and generation of data-tables for plots.

All inversion functions call timer start and stop functions. Timer objects, one per recursion level, are maintained in a list with an iterator to the current level. That way is kept track of which start or stop call belongs to which level and timings for each level are generated.

The constructor of `matrix_inverse` stores the inversion method string and the `matrix_generator` stores the size, condition and the random seed of the matrix. If used, the Newton approximations fill a list of iteration counts. After the inversion, the error measure is stored in the `matrix_statistic` object. With the random seed, it is possible to reproduce to exact same input if exceptional behavior shall be further investigated later.

4.7 Automatic Tuning

As described in section 4.4, the minimal size for parallelization and for recursive transposition is set by an automatically adjusted preprocessor macro. To find values for those parameters, series of automated tests where run that try to narrow the possible range of the optimal value by binary search. They compile a test program with the possible values and compare the runtime of predefined tests. This builds on the assumption, that the gain of parallelization grows monotone with the size. When the lower and upper limit get within a factor of two, setting a value in between no longer makes a difference because OPT's recursions always halve their input size. Then a different input size has to be chosen and two tests to be run with that size and different tuning parameters. The results of these series of tests where so stable and plausible, that in the end the tuning process could be completely automated. The found values are automatically written to an include-file and used in future compiles.

4.8 Possible Improvements

Newton approximation starts off with a very rough approximation and iteratively improves it. Theoretically it takes only one iteration to double the number of correct bits. One could start the approximation with single precision floating point arithmetic and switch to double precision when approaching the end (compare figure 6). Doing so would require conversion of the input and once of an intermediate result, but speed up most of the multiplications by a factor of two plus some advantage due to lower cache footprint. However, OPT only uses Newton approximation on relatively small submatrices, so the gain in that application is limited.

As described in section 4.3, exploiting symmetric multiplications can save almost one fourth of the effort. However, adapting a highly tuned multiplication subroutine to support that feature is not trivial. One has to take into consideration, that for a submatrix on the diagonal not all elements have to be calculated by multiplication. That may either waste valuable cache or it complicates the aligning to cache-line-boundaries if the other elements are not to be stored at their usual places. Also, the rows and columns of the input are not used equally often, thus are different valuable to keep cached. Likewise, parallelization gets more difficult. For full multiplications, splitting the in- and output matrices in parts of equal size results in subtasks of equal size. For symmetric multiplications, such splitting results in tasks of different sizes, which additionally contain to copy the transpose of some parts of the result. Separating multiplication and transposition may help with locality during the multiplication but on the other hand requires reloading the results for the transposition.

Further benefits in parallelization may come from task scheduling. One would define (recursive) inversions and multiplications as tasks that can be split into subtasks by Strassen's algorithm and some recursive multiplication scheme, respectively. That can, for example, be beneficial in the following way: Strassen's inversion algorithm starts with a recursive inversion of the upper left partition. This recursive inversion can begin, as soon as the multiplication that produces the input for the whole inversion finished to compute this partition. The rest of the multiplication can proceed in parallel. In addition to such overlapping, simple parts like copying a matrix can be moved into the multiplication that generates it or that uses it, saving one access to slower memory levels. While the recursion is broken down further, the data-flow comes closer to a streaming mechanism.

Breaking down tasks is not always ambiguous. For example, consider the following multiplication:

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

Breaking it down recursively yields, among other tasks, to compute

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21} \ .$$

If two cores are available, one would compute each product on one processor and then add them afterwards. However, on one core, one could compute the first product and then use the multiply-and-add function of the muladd processor-instruction to add up the second product as it is computed, saving the addition step. Thus, depending on the available parallelism in processing power, the task of multiplication should be broken down differently.

5 Experiments

Main goal of the experiments was to verify the theoretical results claiming practical usability of OPT in that it neither needs too much runtime nor causes too large numerical errors. Theory alone does not suffice in these points, because it neglects cache effects and other factors that can easily increase runtime by magnitudes of hundreds or thousands. Also, while the stability calculation uses rather loose estimates, it still neglects higher order error terms and errors by matrix multiplications that add to errors from recursion. The runtime on different numbers of cores is compared, to examine the scaleability of the algorithms, as well as for different input sizes to see what size is necessary to reach said scaleability.

With additional tests, I examine the coherence between the recursion depth of OPT and the resulting runtime.

I compare the numerical errors of the inverses produced by OPT, OPT-S, Newton approximation, and the MKL inversion routine. During that I observe unexpected behavior of Newton approximation and do some further testing to widen the view on this behavior.

5.1 Random Matrix Generation

As input for the tests, symmetric positive definite matrices are randomly generated to a specified condition. This is possible, because every symmetric positive definite matrix M can be decomposed as $M = QDQ^T$ where Q is an orthogonal matrix and D is a diagonal matrix containing the eigenvalues. The distribution of the eigenvalues determines the condition of the matrix. On the other hand, every matrix composed according to above scheme is symmetric positive definite (see lemma A.3). By choosing Q uniformly distributed, M is chosen uniformly distributed among all symmetric positive definite matrices with the eigenvalues of D.

5.1.1 Randomized Eigenvalue Selection

The difficulty to invert a symmetric positive definite matrix M does not depend on the difference but on the ratio of its eigenvalues. Furthermore, this dependence is not linear but logarithmic. Both manifest in the number of necessary Newton iterations, that include a $\log \frac{\Lambda}{\lambda}$ summand, where Λ and λ are the largest rsp. smallest eigenvalue of M (see theorem 2.10 and lemma 2.4).

As an illustrational example, consider a matrix M with a large and a small eigenvalue Λ' rsp. λ' and a vector $\mathbf{x}$ with its defective floating point representation $\mathbf{x} + \delta_{\mathbf{x}}$. Then $\log \frac{\|\mathbf{x}\|}{\|\delta_{\mathbf{x}}\|}$ gives the approximate number of correct bits in the representation of $\mathbf{x}$. In the worst case $\mathbf{x}$ is an eigenvector to λ', but $\delta_{\mathbf{x}}$ is an eigenvector to Λ'. The number of correct bits in the representation of $M\mathbf{x}$ is given approximately by

$$\log \frac{\|M\mathbf{x}\|}{\|M\delta_{\mathbf{x}}\|} = \log \frac{\lambda'\|\mathbf{x}\|}{\Lambda'\|\delta_{\mathbf{x}}\|} = \log \frac{\|\mathbf{x}\|}{\|\delta_{\mathbf{x}}\|} - \log \frac{\Lambda'}{\lambda'} \ .$$

We see, that the number of correct bits of the representation is reduced by

$$\log \frac{\Lambda'}{\lambda'} = \log \operatorname{cond} M \ .$$

Setting the condition of M is easy: Simply choose an arbitrary λ and $\Lambda = \lambda \cdot \operatorname{cond} M$. It remains to choose the other eigenvalues.

Randomly selecting eigenvalues $\lambda_i \in [1, \operatorname{cond} M]$ uniformly distributed does not have the desired effect of an even distribution on the difficulty: Pick two eigenvalues and let λ_a be the smaller and λ_b the larger. Then with probability $p \geq \left(1 - \frac{\ell-1}{\operatorname{cond} M - 1}\right)^2$ it is $\lambda_a \geq \ell$ and thus $\frac{\lambda_b}{\lambda_a} \leq \frac{\operatorname{cond} M}{\ell}$.[11] For low $\ell \geq 2$, p is high, but the difficulty is reduced significantly.

To follow the log-ratio relation of eigenvalues and difficulty, I selected eigenvalues $\lambda_i = 2^{X_i}$ with X_i chosen randomly uniformly distributed in the interval $[-\frac{1}{2}\log \operatorname{cond} M, \frac{1}{2}\log \operatorname{cond} M]$. This guarantees $\lambda_i \in [\frac{1}{\sqrt{\operatorname{cond} M}}, \sqrt{\operatorname{cond} M}]$. The log-ratio of two eigenvalues is then $\log \frac{\lambda_a}{\lambda_b} = X_a - X_b$. In the implementation I

[11]Actually, the formula given for p is the probability of both λ_a and λ_b to be that large, which is even smaller than p.

skipped explicitly setting largest and smallest eigenvalues, since with high probability the condition is almost at its maximum anyway and I avoid a possible source of undesired asymmetry in M.

5.1.2 Random Orthogonal Matrix Generation

To generate Q, first a matrix X is filled with random entries according to a normal distribution $\mathcal{N}(0,1)$. Then, X is decomposed into an orthogonal and an upper triangular matrix $Q'R' = X$. Last, the diagonal entries of R' are made positive, obtaining R and Q. This can be done by multiplying row i of R' and column i of Q' with $\zeta_i = \operatorname{sign} r'_{ii}$. The result is

$$QR = (Q'\mathrm{Z})(\mathrm{Z}R') = X \quad \text{with } \mathrm{Z} = \begin{pmatrix} \zeta_1 & & 0 \\ & \ddots & \\ 0 & & \zeta_n \end{pmatrix} .$$

In case any $\zeta_i = 0$, X is singular and the generation starts over.

This algorithm is equivalent to iteratively choosing column i of Q uniformly distributed among all directions in $\mathbb{R}^n$ (stored as column i of X), projecting it into the subspace orthogonal to that spanned by columns 1 through $i-1$, and scaling it to length 1 without changing its direction. I chose to make use of the QR decomposition, because efficient implementations are available.

The resulting distribution of Q is uniform in terms of the Haar-Measure. In particular, multiplying the output with an arbitrary orthogonal matrix does not change the distribution.

5.2 Test Setup

The primary line of tests was run on an eight core UMA computer (PC121) with two Intel Xeon 5345, each with four cores, running on 2.33 GHz. Each core has two floating point units, so the theoretical limit on the floating point operations is 4.66 GFLOP/s per core.

The Linux kernel allows to switch of cores separately while the system is running. I use this feature to compare the performance on different numbers of cores. That way it is impossible for any computations to be moved to other cores. The cores to activate were chosen to be as close together as possible (i.e. first activate all cores on one package) to increase the effects of using more cores[12] and thus be able to observe them more easily.

The OpenMP library used for parallelization was instructed to distribute threads evenly and to inhibit moving them between cores. It uses one thread per core by default. For the duration of the tests, no other tasks were run on the computer.

To further investigate the performance on even more cores, a secondary line of tests was run on a larger, 32 core NUMA computer (PCS) with four Intel Xeon X7560, each with 8 cores, running on 2.266 GHz. Each core has two floating point units, so the theoretical limit on the floating point operations is 4.532 GFLOP/s per core. While every socket has fastest access to its own memory, there is no difference in accessing memory of the other sockets. In

[12]The effects in mind were larger combined cache and slower communication between the furthest away cores.

particular is the slowest access when using three or four sockets not slower than when using only two. No mechanism to switch off cores was available. Instead, the OpenMP library was instructed to use only the desired number of threads and distribute all threads in the same fashion as described above. However, tasks controlled by the operation system may have been moved to other cores.

It was made sure, that enough RAM was available, so that no swapping should have occurred. All tests have been repeated multiple times. On PCS, the hours long running test have been repeated only three times; all shorter tests and all tests on the primary test line at least 30 times. Where not otherwise indicated, the results shown are medians of all iterations.

My own code and the includes from Boost were compiled with GNU gcc version 4.4.3 with maximum optimization and static linking. For the matrix multiplication subroutine and as competitor to compare with, it was linked to the appropriate routines of the Intel MKL 10.3 Update 6 which came as part of Intel Composer XE 2011. This is a highly optimized library advertised for use with the processors in use. The linked OpenMP library is also part of Intel Composer XE 2011. I did not make experiments with ATLAS, as compiling it for multi-core computers apparently works not automatically but requires manually "varying `CacheEdge` and iteratively compiling and running `x[pre]l3blastst_pt` until you have a number you are happy with".[18, errata] Therefore I could not expect to get representative results.

All tests use double precision (64-bit) floating point arithmetic. To quantify the error of the results, I use the maximum absolute element of the error matrix, i.e.

$$\text{error}(M, \tilde{M}) = max_{ij}|r_{ij}| \quad \text{with } (r_{ij}) = \mathrm{R}(I - \tilde{M}M) \ . \tag{8}$$

It might be better to compare $\tilde{M}$ with the exact inverse, but that would require to compute the inverse with infinite precision.

5.3 Results

Figure 7 shows the absolute speedup compared to Intel MKL inversion on one core. It can be seen how MKL inversion succeeds to parallelize on up to four cores but struggles to make use of more. Note, that the first four cores are in the same package (compare section 5.2). OPT and OPT-S parallelize without problems, so that on eight cores OPT clearly outperforms MKL inversion.

Comparing OPT on one core with MKL inversion shows that the constant factor on the work in practice is below two, compared to the theoretical four derived in section 3.2.

Figure 8 shows the same test on PCS. The communication between the cores seems to be better, as the MKL inversion makes better use of up to eight cores, possibly because they share a faster higher level cache. Remember, that the first eight cores are in the same package (compare section 5.2). With more than eight cores the performance breaks off, clearly showing the high dependence of the algorithm on good inter-core communication.

Now we examine how well the algorithms utilize the available computation capacity. The total number of floating point operations (FLOP) done by one algorithm depends only on the input size. The numbers for the MKL inversion are taken from the MKL manual,[19] where they are denoted as approximately

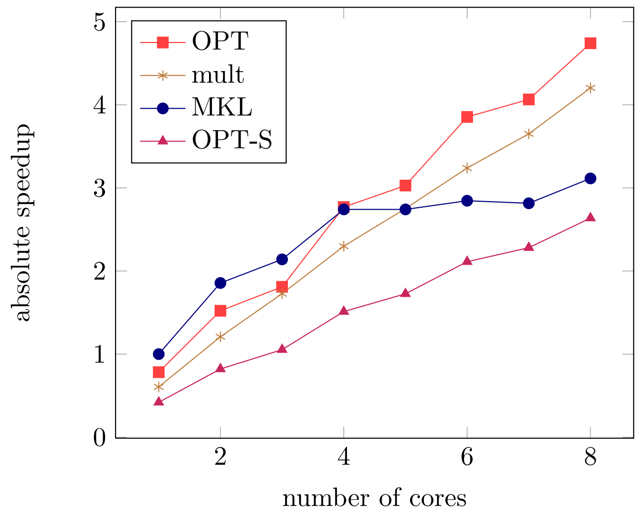

Figure 7: Absolute speedup of inversion methods and multiplication on PC121 compared to Intel MKL inversion on 1 core. Matrix size is 2^{13}.

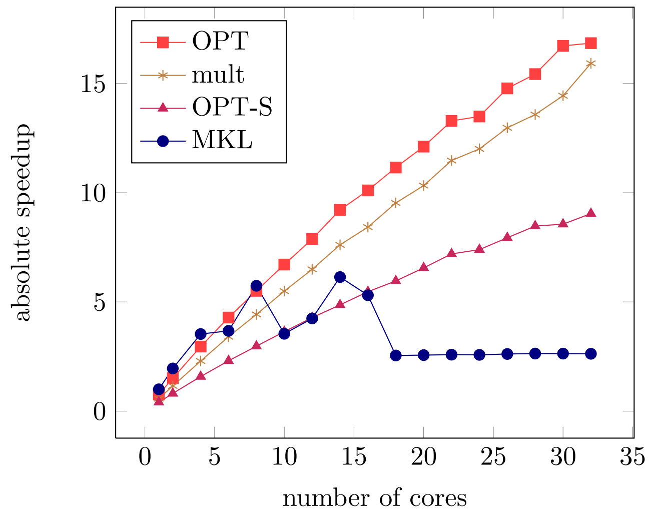

Figure 8: Absolute speedup of inversion methods and multiplication on PCS compared to Intel MKL inversion on 1 core. Matrix size is 2^{14}.

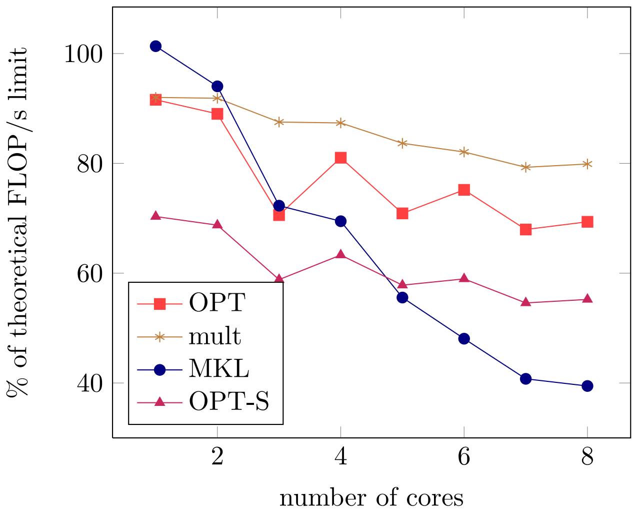

Figure 9: Percentage of the theoretical FLOP/s limit reached by different inversion methods and multiplication on PC121. Matrix size is 2^{13}. The theoretical limit of the computer is 4.66 GFLOP/s per core.
How the value for MKL on one core can be above 100% is discussed in section 5.3.

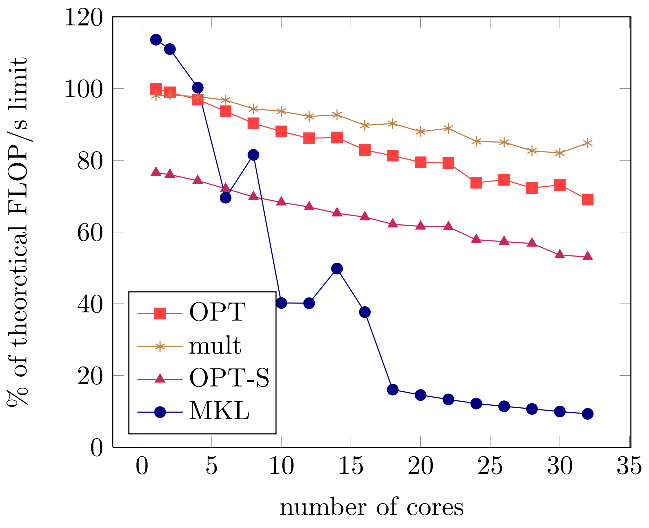

Figure 10: Percentage of the theoretical FLOP/s limit reached by different inversion methods and multiplication on PCS. Matrix size is 2^{14}. The theoretical limit of the computer is 4.532 GFLOP/s per core.
How the value for MKL on one through four cores can be above 100% is discussed in section 5.3.

$\frac{2}{3}n^3$ FLOP. The other numbers are calculated almost exactly, only neglecting some small summands in total in $O(n)$. The numbers are as follows:

	GFLOP	
algorithm	$n = 2^{13}$	$n = 2^{14}$
MKL	366.50	2932.03
Strassen	366.60	2932.43
OPT	423.20	3385.04
OPT-S	605.70	4845.46
multiplication	549.75	4398.04

Division by the theoretical number of FLOP per second (FLOP/s) times the time taken leads to the numbers shown in figures 9 and 10 for PC121 and PCS respectively. Worth noticing is the fact, that the MKL inversion and my implementation of Strassen's algorithm use almost the same amount of FLOP, while my Strassen-implementation does almost one quarter redundant computations (compare section 4.3). That means, that the MKL implementation uses some sub-optimal inversion scheme, too. Additionally observing, that the MKL supposedly achieves more than 100% usage of the floating point units, there are some other interpretations: First, the number in the manual may be exaggerated or rounded up roughly. Second, the MKL might make use of some processor capabilities or optimization possibilities, that are not known to the public. This is quite possible, since the MKL is a highly tuned library made by Intel specifically for their processors. Third, the MKL may simulate some extra floating point operations with integer calculations in the otherwise idle integer ALU. An extreme variant of the first interpretation might be, that the MKL in reality needs only as many operations as an implementation of Strassen's algorithm would need that can make use of the symmetry of some multiplication results. In that case, it would really need only some more than $\frac{1}{2}n^3$ FLOP.

The next test examines the input sizes necessary for effective parallelization. Figure 11 shows that starting from $n = 2^8$ parallelization has a positive effect on the runtime. OPT and OPT-S quickly strive towards the limit of eight, while MKL inversion benefits only little from sizes larger than 2^{10}. Worth noticing is, that MKL multiplication obviously even tries to parallelize on 32×32 matrices, where the effort alone is much more expensive than the whole multiplication.

Again, figure 12 shows the same test on PCS. The picture is very similar with about double sizes, except that MKL inversion does not reach a speedup between three and four but only about 2.6. However, taken from figure 8 a speedup of about six would probably be possible if the parallelization was properly limited to eight cores on this computer.

To examine the numerical stability of OPT and OPT-S, I compare the error of the output as defined by equation (8) with MKL inversion. Figure 13 shows the results. It can be seen, that for not very high conditions (up to 2^{12}) OPT is even better than MKL inversion. Against all expectations and common knowledge, Newton approximation gives the worst results. That realized, it is no wonder that OPT-S produces worse results than OPT, since its intermediate results are worsened by the Newton iterations.

OPT performs $\lceil \log \log n \rceil$ levels of Strassen's algorithm before it reaches the base case. As discussed in section 3, the levels of Strassen's algorithm serve to parallelize most of the large inversion while Newton iterations serve

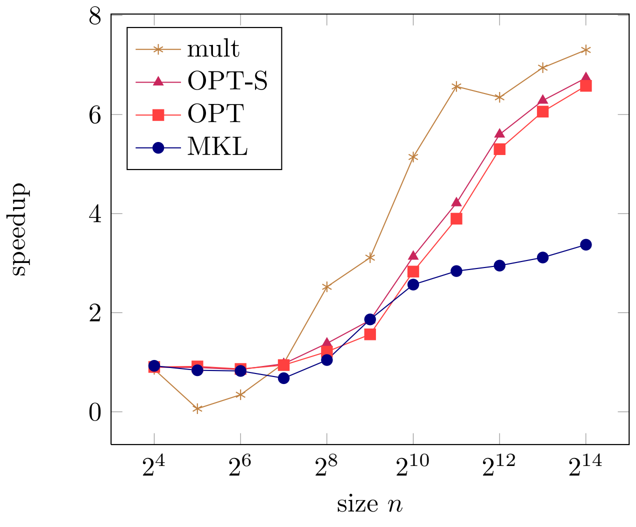

Figure 11: Speedup of inversion methods and multiplication on PC121 with 8 cores compared to the same algorithm on 1 core.

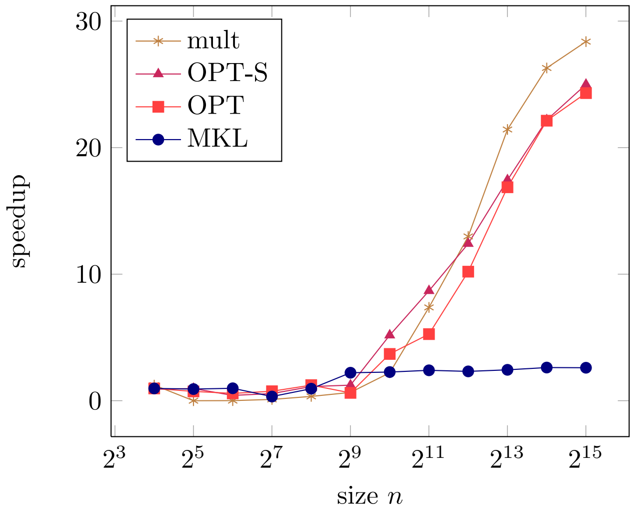

Figure 12: Speedup of inversion methods and multiplication on PCS with 32 cores compared to the same algorithm on 1 core.

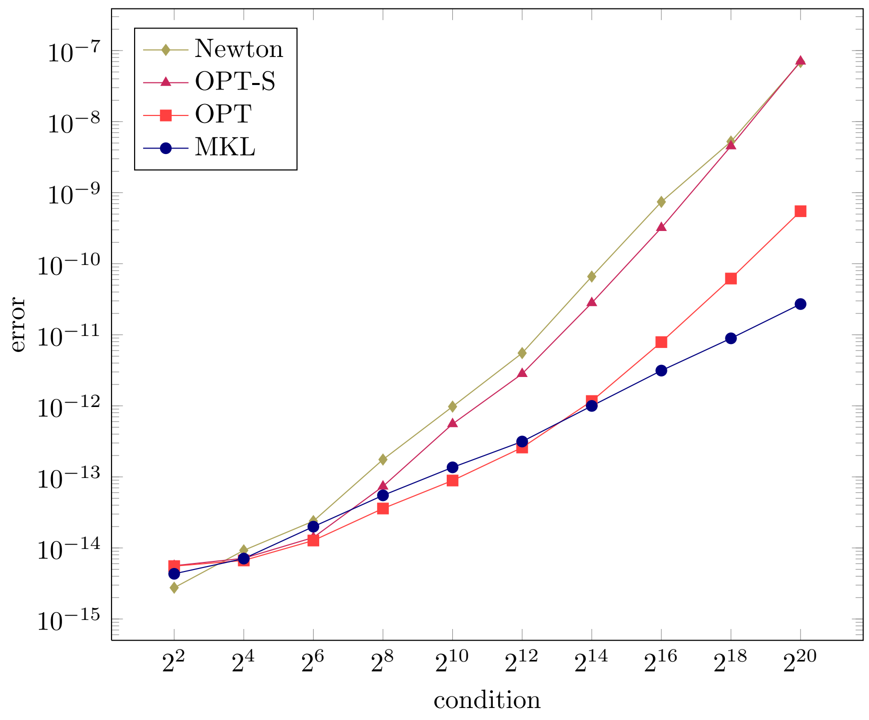

Figure 13: Error (by equation (8)) of different inversion methods by matrix condition for matrices of size 2^{13} and double precision (64-bit) floating point elements. Matrix size is 2^{13}. Calculation was done on PC121.

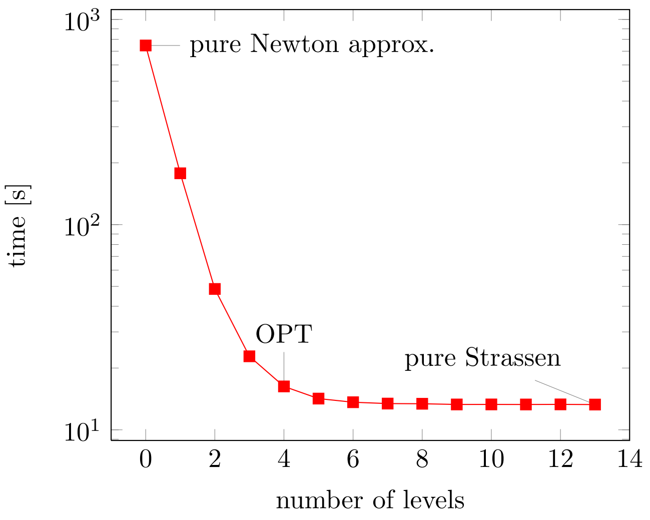

Figure 14: Runtime of OPT altered to do different numbers of levels of recursion. Matrix size is 2^{13}. The number of cores is 8.
0 iterations equal pure Newton approximation, 4 iterations equal OPT, and 13 iterations equal pure Strassen's inversion algorithm.

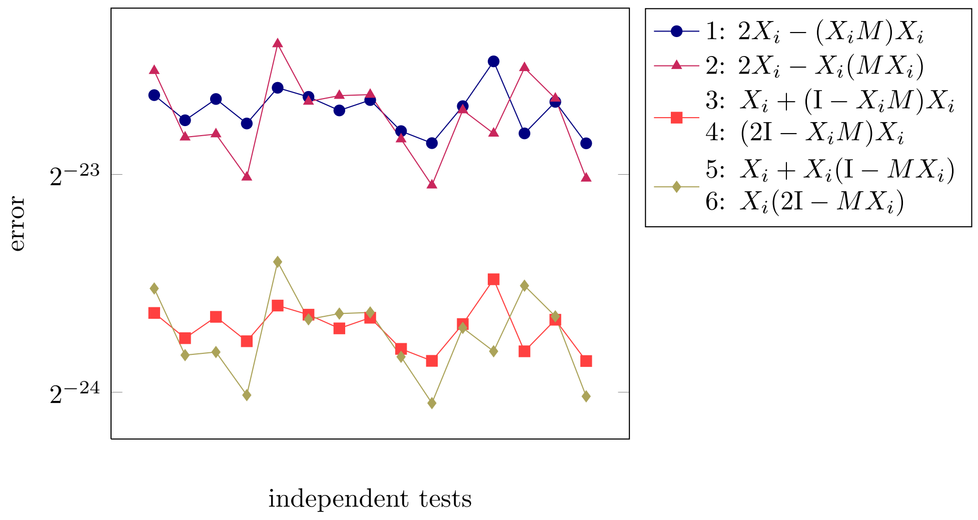

Figure 15: Error of one iteration of Newton approximation after MKL inversion. Matrix size is 2^{13}.
Shown are results of single tests, not medians. On the x-axis are a series of 15 independent tests with different random inputs. Values with the same x-value result from the same input matrix. The connecting lines in this plot indicate no continuity but only serve to better visualize similarities between curves. The values of variants 3 and 4 as well as 5 and 6 are different but so close together, that they would be indistinguishable in the plot.

to parallelize the small base inversions. The boundary of $\log \log n$ fits well for the theory of work and parallel time. The test shown in figure 14 observes the actual runtime for different numbers of levels. It can be seen, that some levels of Strassen's algorithm are necessary to avoid the $\log n$-factor more work of Newton approximation. On an eight core computer, however, the gain by the parallelization of Newton iterations is not visible compared to the much larger amount of work that happens in the outer levels of recursion.

5.4 Instability of Newton Approximation

To investigate the unexpected behavior of Newton approximation, I tried rewriting the formula for one iteration (equation (3)) using associative and distributive laws. I tried the following six variants.

$$\begin{aligned}
1: \quad X_{i+1} &= 2X_i - (X_iM)X_i \\
2: \quad &= 2X_i - X_i(MX_i) \\
3: \quad &= X_i + (\mathrm{I} - X_iM)X_i \\
4: \quad &= (2\mathrm{I} - X_iM)X_i \\
5: \quad &= X_i + X_i(\mathrm{I} - MX_i) \\
6: \quad &= X_i(2\mathrm{I} - MX_i)
\end{aligned}$$

Figure 15 shows errors measured after one iteration of Newton approximation has been applied to the (way less erroneous) result of MKL inversion. I used

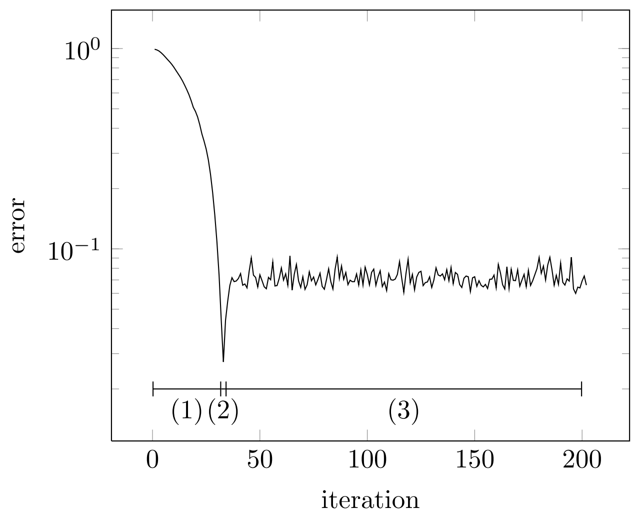

Figure 16: Example of the evolution of the error value during Newton approximation of a very ill conditioned matrix.
Indicated at the bottom are the three phases of evolution of the error (compare section 5.4).

MKL inversion as initial approximation and applied only one iteration to speed up the tests. In a few instances I compared the errors to running a full Newton approximation and found only minor differences.

The similarity of the curves of 1 and 3/4 as well as 2 and 5/6 shows that the (supposed) associativity of X_iMX_i has great influence on the error and, on the other hand, suggests that these multiplications are an important source of error. Compared to the others, variants 1 and 2 produce almost exactly twice the error. These are the variants that double X_i and then subtract X_iMX_i, while the others apply the correction directly to X_i.

My hypothesis is, that the error happens in the first multiplication. X_iM is supposed to yield almost the unity matrix, with very small differences ς_{ij} that make up the correction. Outside of the main diagonal, the full range of the floating point variable is available to store ς_{ij}. Not so inside the main diagonal: Here, $1+\varsigma_{ii}$ is stored in the variable, which leaves only the range of the mantissa to store ς_{ii}. Thus, cancellation occurs on the main diagonal much more than outside of it.

One more observation: The curves of variants 2 and 5/6 look more jittery than those of 1 and 3/4. The multiplication in the error measure is the same as the first multiplication of variants 1 and 3/4.

Figure 16 shows the evolution of the error during the Newton approximation of a very ill conditioned matrix.

The evolution can be divided into three phases: First, the error converges quadratically as expected. Second, the point with minimal error is reached. Third, the supposed correction actually worsens the approximation. The approximations stay in an area where the correction matrix is always erroneous and quadratic improvement never occurs again. Instead, the error fluctuates in a range that is notably worse than the error reached in the second phase.

The behavior in phase three indicates, that the remaining error is not due to a limitation of the Newton formula that prohibits calculation of a better approximation, but that the error matrix can not be computed to the necessary precision. By that i mean, that $(\mathrm{I} - X_i M)$ can be represented with high enough precision to achieve further improvements but $X_i M$ can not, and thus the computation of $(\mathrm{I} - X_i M)$ makes it erroneous.

For better conditioned matrices, the errors in general are smaller and the range of phase three is closer to the minimal error in phase two. Also, phase two becomes longer: The improvement stagnates and approaches the minimal error over multiple iterations before the first worsening. Phase two and three can therefore not always be divided clearly.

6 Summary

With OPT, I presented a technique to create a new algorithm that combines the benefits in work-complexity and parallel time of two others. The key properties of the problem and the existing algorithms where:

- The work grows at least quadratically with the problem size: $W_I(n) \geq n^2$
- Strassen's inversion algorithm
  - is work optimal.
  - is recursive with a constant number of recursive calls.[13]
  - The recursion divides the problem size by a constant factor in each level.[13]
  - Each of the other sub-operations is polylog time parallelizable.
  - does not need to be polylog time parallelizable itself.
- Newton approximation is a polylog parallel time algorithm but causes an additional factor of $\log n$ work compared to a work-optimal algorithm: $W_N(n) = \log n \cdot W_I(n)$

By unrolling the recursion for only $\log \log n$ levels, there are only $\Theta(\log n)$ recursive calls in total and the remaining problem size is in $\Theta(\frac{n}{\log n})$. Since there are only $\Theta(\log n)$ recursive calls, there are as many calls to sub-operations.

Two properties follow for the combined algorithm:

- With the polylog parallel time of Newton approximations, it is a polylog parallel time algorithm, too.
- The $\Theta(\log n)$ calls of size $\Theta(\frac{n}{\log n})$ to Newton approximation cause work in (all in Theta-calculus)

  $$\log n \cdot \log \frac{n}{\log n} \cdot W_I(\frac{n}{\log n}) \leq \log^2 n \cdot W_I(\frac{n}{\log n}) \leq W_I(n) \ .$$

  Since the other operations done by Strassen's algorithm are less than when the recursion is unrolled completely, both parts of work and thus the whole algorithm are work-optimal.

[13] Both constants are two, but that is less general and not necessary for the reasoning.

In sections 2.1.2 and 2.1.3 we have seen, how an algebraic technique is superior in a numerical sense to classic pivoting and swapping. It remains to note, that by its data-oblivious it additionally provides a predictable instruction and data flow. This property should be especially useful for fine-grained task-scheduling and cache prefetching.

In the experiments about numerical stability I stumbled on an unexpected instability of Newton approximation. Strassen's algorithm, on the other hand, gave fine results.

6.1 Future Work

The technique resulting in OPT may be applicable in a wide area of algorithms. In the closer area of linear algebra, it would be interesting to try and apply it to related tasks such as LU and Cholesky decomposition and linear systems solving. Recursive algorithms for those decompositions have already been presented, e.g. by Reif.[9]

The surprising instability of Newton approximation raises the question of either improvement of its calculations or a different practical polylog time algorithm that does not need to be work-optimal.

Although OPT is not a blocked algorithm, the recursion and the many large multiplications can be interpreted to generate many small tasks, too. Those tasks can then be scheduled with techniques that are current interest of research, similar as with Netlib's LAPACK implementation that builds on MPI for parallelization.[4] I described the possible benefits and complications of task-scheduling to OPT in section 4.8.

A Symmetric Positive Definite Matrices and Condition

Lemma A.1.

Let $M \in \mathbb{R}^{n \times n}$ *non-singular*

Then $M^T M$ *is symmetric positive definite*

Proof.
$M^T M$ is trivially symmetric.

Let $\mathbf{x} \in \mathbb{R}^n \setminus 0$

Then $\mathbf{x}^T M^T \underbrace{M\mathbf{x}}_{\neq 0} = \|M\mathbf{x}\|^2 > 0$

□

Lemma A.2.

Let $M \in \mathbb{R}^{n \times n}$ *non-singular*

Then $\operatorname{cond}(M^T M) \leq \operatorname{cond}^2 M$

Proof.

$$\|M^T M\| \leq \|M^T\| \cdot \|M\|$$
$$\|(M^T M)^{-1}\| = \|M^{-1} M^{-T}\| \leq \|M^{-1}\| \cdot \|M^{-T}\|$$

□

Lemma A.3.

Let $Q \in \mathbb{R}^{n \times n}$ *orthogonal (i.e.* $Q^T = Q^{-1}$*)*

$D \in \mathbb{R}^{n \times n}$ *diagonal with positive diagonal entries* d_{ii}

Then QDQ^T *is symmetric positive definite*

Proof.

QDQ^T is symmetric since D is symmetric.

Let $\mathbf{x} \in \mathbb{R}^n \setminus 0$

$$\mathbf{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix} = Q^T \mathbf{x}$$

Then $\mathbf{y} \neq 0$ since Q is non-singular

$$\mathbf{x}^T QDQ^T \mathbf{x} = \mathbf{y}^T D \mathbf{y} = \sum_{i=1}^{n} d_{ii} y_i^2 > 0$$

Thus QDQ^T is positive definite.

□

References

[1] Virginia Vassilevska Williams: Breaking the Coppersmith-Winograd Barrier. available at: <http://www.cs.berkeley.edu/ virgi/matrixmult.pdf>. 2011.

[2] Jakub Kurzak, Hatem Ltaief, Jack Dongarra, Rosa M. Badia: Scheduling Dense Linear Algebra Operations on Multicore Processors. *Concurrency and Computation: Practice and Experience, Volume 22, Issue 1 (2010), ISSN: 1532-0634, DOI: 10.1002/cpe.1467.* Wiley, 2009/2010.

[3] Fengguang Song, Asim YarKhan, Jack Dongarra: Dynamic Task Scheduling for Linear Algebra Algorithms on Distributed-Memory Multicore Systems. *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC '09), DOI: 10.1145/1654059.1654079.* ACM, New York, NY, USA, 2009.

[4] Alfredo Buttari, Julien Langou, Jakub Kurzak, Jack Dongarra: A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures. *Parallel Computing, Volume 35, Issue 1, January 2009, Pages 38-53, ISSN 0167-8191, 10.1016/j.parco.2008.10.002.* Elsevier B.V., 2009.

[5] James Demmel, Ioana Dumitriu, Olga Holtz: Fast Linear Algebra is Stable. *Numerische Mathematik, Volume 108, Issue 1 (2007), Pages 59-91, ISSN: 0029-599X, DOI: 10.1007/s00211-007-0114-x.* Springer, Berlin / Heidelberg, 2007.

[6] S. Chatterjee, A. R. Lebeck, P. K. Patnala, and M. Thottethodi: Recursive Array Layouts and Fast Matrix Multiplication. *IEEE Transactions on Parallel and Distributed Systems, Volume 13, Number 11 (2002), Pages 1105-1123, ISSN: 1045-9219, DOI: 10.1109/TPDS.2002.1058095.* IEEE Computer Society Press, Los Alamitos, CA, USA, 2002.

[7] B. Codenotti, M. Leoncini, F. P. Preparata: The Role of Arithmetic in Fast Parallel Matrix Inversion. *Algorithmica, Volume 30, Number 4 (2001), Pages 685-707, ISSN: 0178-4617, DOI: 10.1007/s00453-001-0033-7.* Springer, New York, 2001.

[8] G. H. Golub, C. F. Van Loan: Matrix computations (3rd edition). ISBN: 0-8018-5414-8. Johns Hopkins University Press, Baltimore, 1996.

[9] John H. Reif: O(log^2n) Time Efficient Parallel Factorization of Dense, Sparse Separable, and Banded Matrices. *Proceedings of the sixth annual ACM symposium on Parallel algorithms and architectures (SPAA '94), Pages 278-289, DOI: 10.1145/181014.181408.* ACM, New York, NY, USA, 1994.

[10] Victor Pan, John Reif: Fast and efficient parallel solution of dense linear systems. *Computers & Mathematics with Applications, Volume 17, Issue 11, 1989, Pages 1481-1491, ISSN 0898-1221, 10.1016/0898-1221(89)90081-3.* Pergamon Press plc, 1989.

[11] D. H. Bailey, H. R. P. Ferguson: A Strassen-Newton algorithm for high-speed parallelizable matrix inversion. *Proceedings of the 1988 ACM/IEEE conference on Supercomputing (Supercomputing '88), Pages 419-424.* IEEE Computer Society Press, Los Alamitos, CA, USA, 1988.

[12] Victor Pan, John Reif: Efficient Parallel Solution of Linear Systems. *Proceedings of the seventeenth annual ACM symposium on Theory of computing (STOC '85).* ACM, New York, NY, USA, 1985.

[13] L. Csanky: Fast Parallel Matrix Inversion Algorithms. *SIAM Journal on Computing, Volume 5, Number 4 (1976), Pages 618-623, ISSN: 0097-5397, DOI: 10.1137/0205040.* SIAM, Philadelphia, USA, 1976.

[14] L. Csanky: On the Parallel Complexity of Some Computational Problems. Ph.D. dissertation. Computer Science Division, University of California, Berkley, 1974.

[15] Volker Strassen: Gaussian Elimination is Not Optimal. *Numerische Mathematik, Volume 13, Issue 4, 1969, Pages 354-356, ISSN 0029-599X, DOI: 10.1007/BF02165411.* Springer, Berlin/Heidelberg, 1969.

[16] Basic Linear Algebra Subprograms (BLAS). original publication of the definition available at: <http://www.netlib.org/blas/>. Netlib.

[17] Linear Algebra PACKage (LAPACK). original publication of the definition and source code available at: <http://www.netlib.org/lapack/>. Netlib.

[18] Automatically Tuned Linear Algebra Software (ATLAS). used version: 3.8.4 available at: <http://math-atlas.sourceforge.net/>. Netlib.

[19] Intel Math Kernel Library Reference Manual. Document Number: 630813-043US, available at: <http://software.intel.com/sites/products/documentation/hpc/composerxe/en-us/2011Update/mklxe/mkl_manual_lnx/index.htm>. Intel Corporation, 2011.

[20] Boost C++ Librarys. used version: 1.48.0 available at: <www.boost.org>.