Bachelor Thesis

Communication-Avoiding Parallel Algorithms for Solving Triangular Matrix Equations

Author(s):
Wicky, Tobias

Publication Date:
2015

Permanent Link:

https://doi.org/10.3929/ethz-a-010686133 →

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Communication-Avoiding Parallel Algorithms for Solving Triangular Matrix Equations

Bachelor Thesis

Tobias Wicky

Friday 30th October, 2015

Advisors: Prof. Dr. T. Hoefler, Dr. E. Solomonik
Department of Mathematics, ETH Zürich

Abstract

In this work an algorithm for solving triangular systems of equations for multiple right hand sides is presented. The algorithm for solving triangular systems for multiple right hand sides, commonly referred to as the TRSM problem, is a very important in dense linear algebra as it is a subroutine for most decompositions of matrices as LU or QR. To improve performance over the standard iterative algorithms for TRSM, a block wise inversion paired with triangular matrix multiplications is used. To perform the inversion, the lower triangular form of the matrix is exploited and a recursive scheme is applied to further decrease communication cost. With that, the latency of the algorithm decreases while the bandwidth and floating point operations count stay asymptotically the same. Concretely, a decrease of latency with a factor of $p^{2/3} / \log p$ was achieved for a significant range of relative matrix sizes when working with p processors. The proposed method is implemented and its performance is benchmarked against the widely used ScaLAPACK [1] library. The results show promising tendencies for the inversion, with a maximal speedup of 1.7 over ScaLAPACK for 4096 processors. Due to the inferior performance of triangular matrix multiplications with respect to the triangular solve, no overall improvement is made yet.

Contents

Chapter 1

Introduction

The goal of this work is to find a communication minimizing algorithm for solving triangular matrix equations with multiple right hand sides.

Motivation: With the decreasing importance of the amount of floating point operations on the total execution time of a program, communication costs will turn out to be a more and more important factor. The goal therefore is to find an algorithm that stays asymptotically optimal in terms of flop-cost while decreasing the communication cost. The approach taken here is to specifically decrease the asymptotic latency cost while keeping the bandwidth constant. We take this approach because we assume that the asymptotic upper bounds for the bandwidth are very strong for the standard recursive approach as described in [2] since they are equal to the bandwidth costs that are required in for matrix multiplication. TRSM is a crucial problem in many applications as it is a subroutine for a lot of algorithms in dense linear algebra, as for example the LU decomposition described in [3], or the QR decomposition. An other very important application of the TRSM is the recursive Cholesky factorization where it is the base-case algorithm as described in [4]. It is also the critical routine for solving general dense linear systems of equations.

Problem Definition: The linear system of equations

$$L \cdot X = B$$

will be solved for $X \in \mathbb{R}^{n \times k}$, where $L \in \mathbb{R}^{n \times n}$ denotes a lower-triangular matrix and $B \in \mathbb{R}^{n \times k}$ a dense matrix. In this work we will only account for communication cost (bandwidth and latency) that arises due to communication between different compute units. In the theoretical part of this work, three algorithms are presented to solve the problem recursively where each algorithm takes a different approach: The algorithm referred to as **Recursive TRSM** splits the initial problem into smaller subproblems, until the problem

is small enough such that each compute unit can solve the given subproblem for some different right hand sides. This first approach is based on the approach taken in [2]. With a detailed cost analysis we will describe optimal base-case sizes dependent on the relative matrix sizes. This approach is explained in detail in Section 3.2. The algorithm referred to as **TRSM with full inversion** inverts the triangular matrix completely in a recursive fashion and then solves the system by a triangular multiplication of the inverted matrix L^{-1} with the right hand side B. This approach is discussed due to the low latency that is required to do inversion and matrix multiplication. In some relative matrix sizes this approach provides an overall higher bandwidth cost and floating point operations count on the cost of the low latency. It is discussed in detail in Section 5.1. The algorithm referred to as **Recursive-Inversion TRSM** combines the two approaches, reducing the problem recursively up until a certain point and inverts the small problem to solve it. With this approach we aim at a lower latency due to the use of the inversion as well as keeping the bandwidth and floating point operations count low as we can choose the base case size as it is desired. With a cost analysis of this approach, we were able to find a cost optimal base-case size that decreases the latency and keeps bandwidth as well as flop-costs constant compared to the recursive TRSM. With p processors working on the problem, the decrease of latency was obtained for a large range of relative matrix sizes $\left(k \in \left[\sqrt{p}, np\right)\right)$ where a gain of a factor of $\frac{p^{2/3}}{\log p}$ was achieved. It is discussed in Section 5.2.

Results: To see how the algorithm performs in practice we show scaling plots of the Recursive-Inversion TRSM compared to the method that ScaLAPACK [1] provides. We were able to see that our approach of inverting the lower triangular matrix is faster than what ScaLAPACK provides. Even though this is an important part of the algorithm, we observed worse scaling for the total time to solution for TRSM. This can be explained by the fact that one very slow part of the algorithm was the triangular matrix multiplication that uses ScaLAPACKs implementation. The plots suggest a good scaling behavior but they show that the problem size used was very small and that with increasing problem sizes, better results are expected.

Chapter 2

Previous Work

In this chapter, the previous work relevant to the topic is introduced: A cost analysis for matrix multiplication is presented and a standard method for solving the TRSM problem is shown.

2.1 Matrix Multiplication

In this section the relevant results from the CARMA algorithm, including the cost analysis for matrix multiplication, presented in [5], are summarized.

One of the key parts of recursive algorithms for solving the TRSM problem are matrix matrix multiplications. In [5], Demmel et al present communication optimal algorithms for matrix multiplications with the respective costs. The algorithm we present is based on the matrix multiplication presented in their work, where a threefold case distinction occurs. The fact that these three different cases all have different bounds for communication costs sets the constraint that the algorithm presented here also has to do the same distinction of cases.

Initial conditions: We consider the matrix multiplication of a dense matrix $A \in \mathbb{R}^{n \times n}$ with an other dense matrix $B \in \mathbb{R}^{n \times k}$ executed on p processors that are aligned on a processor gird Π.

Bandwidth: The CARMA algorithm [5] which we refer to as $C = \mathrm{MM}(A, B, n, k, \Pi, p)$ achieves communication bandwidth cost $W_{\mathrm{MM}}(n, k, p)$, which can be subdivided into three cases:

$$W_{\mathrm{MM}}(n,k,p) = \begin{cases} \mathcal{O}\left(nk/\sqrt{p}\right) & n > k \cdot \sqrt{p} \quad \text{(two large dimensions)} \\ \mathcal{O}\left(\left(\frac{n^2 k}{p}\right)^{2/3}\right) & k/p \leq n \leq k \cdot \sqrt{p} \quad \text{(three large dimensions)} \\ \mathcal{O}\left(n^2\right) & n < k/p \quad \text{(one large dimension)} \end{cases}$$

In the case of one large dimension, where the right hand side B is larger than the triangular matrix A, the best way to do a matrix multiplication is to use a one dimensional layout for the processor grid.

In the case of two large dimensions, where the matrix A is much larger than the right hand side B, the best way of performing a matrix multiplication is to use a two dimensional layout for the processor grid.

And for the case of three large dimensions, where the matrices A and B are approximately of the same size, it is proposed to use a three dimensional grid layout.

Latency: The latency cost of matrix multiplication given unlimited memory is

$$S_{\text{MM}}(n,k,p) = \log(p)$$

Flop Cost: Matrix multiplication takes $\mathcal{O}\left(n^2k\right)$ flops, which can be divided on p processors and therefore we have

$$F_{\text{MM}}(n,k,p) = \mathcal{O}\left(\frac{n^2k}{p}\right)$$

Previous Analysis: For the case where $k = n$ the bandwidth analysis of a general matrix multiplication goes back to what is presented in [6]. Aggarwal et al. presented a cost analysis in the LPRAM model. In this work, the authors showed that the same cost can also be achieved for the transitive closure problem that can be extended to the problem of doing an LU decomposition. The the fact that these bandwidth costs can be obtained for the LU decomposition was later demonstrated by Tiskin in [7]. He used the bulk synchronous parallel (BSP) execution time model. Since the dependencies in LU are more complicated than they are for TRSM, we also expected TRSM to be able to have the same asymptotic costs as a general matrix multiplication.

2.2 Triangular matrix solve for single right hand sides

Algorithms for the problem of triangular solve for a single right hand side (when X and B are vectors) have been well-studied. A communication-efficient parallel algorithm was given by Heath and Romine [8]. This parallel algorithm was later shown to be an optimal schedule in latency and bandwidth costs via lower bounds [9]. However, when X and B are matrices ($k > 1$), it is possible to achieve significantly lower communication costs relative to the amount of computation required.

2.3 Triangular matrix solve for multiple right hand sides

The idea of recursively solving triangular matrix systems with many right hand sides already showed up a long time ago in the work of Elmroth et al [2]. There are two main ways to split the problem into smaller subproblems:

Case 1: Splitting the right hand side into two (independent) subproblems in the fashion

$$A \cdot X = B$$

becoming

$$A \cdot \begin{bmatrix} X_1 & X_2 \end{bmatrix} = \begin{bmatrix} B_1 & B_2 \end{bmatrix}$$

where the subproblems

$$A \cdot X_1 = B_1$$
$$A \cdot X_2 = B_2$$

are solved independently.

Case 2: Splitting the triangular matrix into two dependent subtask in the fashion of

$$\begin{bmatrix} A_{11} & \\ A_{12} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} X_1 \\ X_2 \end{bmatrix} = \begin{bmatrix} B_1 \\ B_2 \end{bmatrix}$$

where the subproblems

$$A_{11} \cdot X_1 = B_1$$
$$A_{22} \cdot X_2 = B_2 - A_{12} \cdot X_1$$

are solved. With a proper mixing of both cases and a new approach of calculating the base-cases of these recursions, it was possible to achieve good bandwidth and latency costs.

In [10], it is shown that a sequential execution of the TRSM algorithm can achieve the same bandwidth cost as a general matrix matrix multiplication. This is done using the $\alpha - \beta$ model and accounting for different cache-line sizes. The cost analysis done in [11] shows that bandwidth-wise, one can achieve costs for TRSM that are not worse than what a general matrix multiplication achieves. We do a similar analysis with a different model in Section 3.2.

The stability of inverting a triangular matrix has been studied in [12]. It has been stated that no general error bounds exist for blocked methods with respect to the error bound of the iterative methods. We leave the investigation of the stability of our approach as further work.

Chapter 3

Communication in TRSM

In this chapter we provide a model to calculate parallel execution time and we make a cost analysis of the recursive TRSM algorithm using the communication upper bounds that were presented in the previous chapter.

3.1 Execution Time Model

The model we use to calculate the parallel execution time of an algorithm along its critical path is the $\alpha - \beta - \gamma$ model. It describes the total execution time of the algorithm T in terms of the floating point operations (flop) count F, the bandwidth W and the latency (synchronization cost) S along the critical path:

$$T = \gamma \cdot F + \beta \cdot W + \alpha \cdot S$$

We do not place constraints on the local memory size.

As it is assumed that with time, computing elements will become faster and with that a decrease of γ is expected, the goal of this work is to find an algorithmic approach to solving triangular matrix equations with multiple right hand sides that only increases the flop cost F by a constant, while decreasing the latency S. This is a reasonable approach, since the importance of α and β becomes higher as γ gets lower.

3.2 Recursive TRSM Algorithm

In this section, the algorithmic approach to solving TRSM recursively is presented. Also a cost analysis of the recursive approach of TRSM is shown.

Algorithmic approach: The algorithm to solve many triangular systems of equations, commonly referred to as TRSM problem reads: Given a lower-triangular matrix $L \in \mathbb{R}^{n \times n}$ and a dense matrix $B \in \mathbb{R}^{n \times k}$, the goal is to

compute the matrix $X \in \mathbb{R}^{n \times k}$ such that

$$L \cdot X = B.$$

The recursive algorithm that is presented by Elmroth et al. in [2] subdivides the L matrix into a 2×2 set of blocks at each step and performs two TRSM calls in sequence with all processors at each recursive level. Algorithm 1,

Algorithm 1: $X = \text{Rec-TRSM}(L, B, n, k, \Pi, p, n_0)$

Require: *L is a lower triangular $n \times n$ matrix and B is a rectangular $n \times k$ matrix, both distributed over $|\Pi| = p$ processors.*
If $n \leq n_0$, allgather L onto all processors, subdivide $B = [B_1, \ldots, B_p]$ and $X = [X_1, \ldots, X_p]$ and compute $X_i = L^{-1}B_i$ with the ith processor.
Subdivide L into $n/2 \times n/2$ blocks, $L = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix}$
Subdivide B and X into $n/2 \times k$ blocks, $B = \begin{bmatrix} B_1 \\ B_2 \end{bmatrix}$, $X = \begin{bmatrix} X_1 \\ X_2 \end{bmatrix}$.
Compute $X_1 = \text{Rec-TRSM}(L_{11}, B_1, n/2, k, \Pi, p, n_0)$.
Compute $B_2' = B_2 - \text{MM}(L_{21}, X_1, n/2, k, \Pi, p)$.
Compute $X_2 = \text{Rec-TRSM}(L_{22}, B_2', n/2, k, \Pi, p, n_0)$.
Ensure: $LX = B$.

Rec-TRSM(L, B, n, k, Π, n_0) requires two recursive calls and a matrix multiplication at each recursive level until $n \leq n_0$.

Bandwidth Cost: This approach yields to the communication cost recurrence

$$W_{\text{Rec-TRSM}}(n, k, p, n_0) = W_{\text{MM}}(n/2, k, p) + 2W_{\text{Rec-TRSM}}(n/2, k, p, n_0)$$

which decreases geometrically at each level as long as $k > n/\sqrt{p}$ (first case of Algorithm 1). At the base-case, the all-gather of L requires a communication cost of

$$W_{\text{Rec-TRSM}}(n_0, k, p, n_0) = \mathcal{O}\left(n_0^2\right)$$

There are n/n_0 base-cases that are executed in sequence using all processors, for a total cost of

$$W_{\text{base-cases}}(n, k, p, n_0) = \frac{n}{n_0} \cdot W_{\text{Rec-TRSM}}(n_0, k, p, n_0) = \mathcal{O}\left(nn_0\right)$$

3.2.1 Choice of base-case size

We desire that $W_{\text{base-cases}}(n, k, p, n_0) \leq W_{\text{TRSM}}(n, k, p, n_0)$, which implies that we need a different choice of n_0 depending on the initial size of our matrix:

One large dimension: In this case, because the matrix A is small, it makes no sense to split it and therefore we would pick $n_0 = n$ and with that choice, no recursion occurs.

Two large dimensions: When the initial matrix multiplication costs $\mathcal{O}\left(nk/\sqrt{p}\right)$ and we want the base-case not to dominate, we choose

$$n_0 = \max\left(1, \frac{k}{\sqrt{p}}\right)$$

It is important to note that as we recurse, the matrix multiplications in each level costs the same amount of bandwidth, and therefore we pick up a logarithmic factor for the total bandwidth.

Three large dimensions: When the initial matrix multiplication costs $\mathcal{O}\left(\frac{n^2k}{p}\right)^{2/3}$, we select

$$n_0 = \max\left(1, \left(\frac{nk^2}{p^2}\right)^{1/3}\right)$$

Latency: The latency cost is dominated by the n/n_0 base-cases since they may not be executed concurrently and therefore comprise a execution path within the algorithm of $S_{\text{Rec}-\text{TRSM}}(n,k,p,n_0) = (n/n_0) \cdot S_{\text{MM}}(n_0,k,p)$. Each base-case requires an all-gather, which implies $S_{\text{MM}}(n_0,k,p) = \mathcal{O}\left(\log(p)\right)$ latency cost, yielding an overall latency cost of

$$S_{\text{Rec}-\text{TRSM}}(n,k,p,n_0) = \mathcal{O}\left((n/n_0)\log p\right)$$

This general cost leads to the following latency costs, when the choice of n_0 is made to minimize bandwidth cost with respect to the initial matrix multiplication, as done above.

One large dimension:

$$S_{\text{Rec}-\text{TRSM}}(n,k,p,n) = \mathcal{O}\left(\log p\right)$$

Two large dimensions:

$$S_{\text{Rec}-\text{TRSM}}\left(n,k,p,\frac{k}{\sqrt{p}}\right) = \mathcal{O}\left(\min\left(1,\frac{\sqrt{p}}{k}\right) n\log p\right)$$

Three large dimensions:

$$S_{\text{Rec}-\text{TRSM}}\left(n,k,p,\left(\frac{nk^2}{p^2}\right)^{1/3}\right) = \mathcal{O}\left(\min\left(n,\left(\frac{np}{k}\right)^{2/3}\right)\log p\right)$$

Flop Cost: The flop cost of such an algorithm is dominated by the top level matrix multiplications and therefore costs

$$F_{\text{Rec}-\text{TRSM}}(n,k,p) = \mathcal{O}\left(\frac{n^2k}{p}\right)$$

Chapter 4

Triangular Matrix Inversion

In this chapter, the algorithmic approach to inverting a lower triangular system recursively is presented. Also a cost analysis of this approach is presented.

Algorithmic approach: As the scaling of the latency of the algorithm discussed previously grows with the number of processors involved, its scalability is limited. Therefore, this approach seems suboptimal and with that the goal is to find an algorithm that requires less latency. Noting that matrix multiplication requires little latency, the cost of inverting a triangular matrix was investigated. Parallel triangular matrix inversion can be done with a shorter critical path than the TRSM approach discussed before. This approach is shown in Algorithm 2. It is to note that whenever a call to MM is made, we speak of the matrix multiplication mentioned in [5]. The approach taken in Algorithm 2 is the following: Each problem is subdivided into two recursive matrix inversions, which are executed concurrently with half the processors, then two matrix multiplications are performed to complete the inversion.

Bandwidth cost: Since we execute the two recursive calls in
$\mathrm{Rec}-\mathrm{Tri}-\mathrm{Inv}(L,n,\Pi,p)$ simultaneously with two disjoint sets of processors, the communication cost recurrence is given by

$$W_{\mathrm{Rec-Tri-Inv}}(n,p) = 2 \cdot W_{\mathrm{MM}}(n/2,n/2,p) + W_{\mathrm{Rec-Tri-Inv}}(n/2,p/2,n_0)$$

If we assume n to be sufficiently large, we find that $p = 1$ should be achieved and that is when $n_0 = n/2^{\log p} = n/p$, so therefore there are a total of $\log p$ recursive levels, each of which requires a matrix multiplication. The communication cost associated with each level decreases geometrically, therefore the total cost of the algorithm is dominated by the cost of the top-level matrix multiplication, which is

$$W_{\mathrm{MM}}(n/2,n/2,p) = \mathcal{O}\left(\frac{n^2}{p^{2/3}}\right)$$

Algorithm 2: $L^{-1} = \text{Rec-Tri-Inv}(L, n, \Pi, p)$

Require: *L, a lower triangular $n \times n$ matrix distributed over $|\Pi| = p$ processors.*
if $p = 1$ **then**
  $L^{-1} = \text{sequential inversion}(L)$
end
else
  Subdivide L into $n/2 \times n/2$ blocks, $L = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix}$
  Subdivide $\Pi = [\Pi_1, \Pi_2]$ where Π_1 and Π_2 each contain $p/2$ processors
  $L_{11}^{-1} = \text{Rec-Tri-Inv}(L_{11}, n/2, \Pi_1, p/2)$
  $L_{22}^{-1} = \text{Rec-Tri-Inv}(L_{22}, n/2, \Pi_2, p/2)$
  $L_{21}'^{-1} = -\text{MM}(L_{22}^{-1}, L_{21}, n/2, n/2, \Pi, p)$
  $L_{21}^{-1} = \text{MM}(L_{21}'^{-1}, L_{11}^{-1}, n/2, n/2, \Pi, p)$
  Assemble L^{-1} from the $n/2 \times n/2$ blocks, $L^{-1} = \begin{bmatrix} L_{11}^{-1} & 0 \\ L_{21}^{-1} & L_{22}^{-1} \end{bmatrix}$
end
Ensure: $LL^{-1} = I$.

Since this cost of the matrix multiplication decreases as n and p both are decreased by a factor of two, this initial matrix multiplication dominates the bandwidth cost,

$$W_{\text{Rec-Tri-Inv}}(n, p, n_0) = \mathcal{O}\left(W_{\text{MM}}(n/2, n/2, p)\right) = \mathcal{O}\left(\frac{n^2}{p^{2/3}}\right).$$

Latency cost: Since there are $\log(p)$ recursive levels and at each step we do a matrix-matrix multiplication, the latency cost is

$$S_{\text{Rec-Tri-Inv}}(n, p) = \mathcal{O}\left(\log^2(p)\right)$$

Flop cost: The total flop cost for the inversion is

$$F_{\text{Rec-Tri-Inv}}(n, n_0, p) = F_{\text{Base-Cases}}(n, n_0, p) + F_{\text{MM}}(n/2, n/2, p)$$

The flop cost of a sequential inversion of a triangular matrix is, as stated by Hunger in [13],

$$F_{\text{Seq-Inv}}(n) = \frac{1}{3}n^3$$

and the total base-case flop cost is

$$F_{\text{Base-Cases}}(n, n_0, p) = \frac{n}{n_0} F_{\text{Seq-Inv}}(n_0) = \frac{n}{n_0} \cdot \frac{1}{3} n_0^3 = \frac{1}{3}\left(\frac{n}{p}\right)^3$$

This gets dominated by the top-level matrix multiplication, which every processor is working on and therefore needs

$$F_{\mathrm{MM}}(n/2, n/2) = \mathcal{O}\left(\frac{n^3}{p}\right)$$

flops. Since the top level matrix multiplication is the most expensive and the other levels are geometrically decreasing this gives us the cost of

$$F_{\mathrm{Rec-Tri-Inv}}(n, n_0, p) = \mathcal{O}\left(\frac{n^3}{p}\right)$$

Chapter 5

TRSM with Inversion

In this chapter, we discuss approaches to solve the TRSM problem using the inversion derived in Section 4. The idea to use inversion for its low latency costs arises from what Tiskin did in [7]: He used inversion to decrease the latency in the LU factorization. We discuss optimal base-case choices dependent on relative matrix sizes for both TRSM with full inversion as well as recursive TRSM with block inversion.

5.1 TRSM with full inversion

In this section, the algorithm to solving TRSM with a complete inversion of L is given. We provide a cost analysis for this method as well as an optimal base-case choice.

Algorithmic approach: If TRSM is done with full inversion of the matrix, the algorithm works as described in Algorithm 3.

Algorithm 3: $X = \text{Inv-TRSM}(L, B, n, k, \Pi)$

Require: *L is a lower triangular $n \times n$ matrix and B is a rectangular $n \times k$ matrix, both distributed over $|\Pi| = p$ processors.*
$L^{-1} = \text{Rec-Tri-Inv}(L, n, \Pi)$
$X = \text{MM}(L^{-1}, B, n, k, \Pi)$
Ensure: $LX = B$.

Bandwidth cost: This approach leads to a total bandwidth cost of

$$W_{\text{Inv-TRSM}}(n,k,p,n_0) = W_{\text{Rec-Tri-Inv}}(n,p,n_0) + W_{\text{MM}}(n,k,p)$$
$$= \begin{cases} \mathcal{O}\left(\frac{n^2}{p^{2/3}}\right) & n > k\cdot\sqrt{p} \quad \text{(two large dimensions)} \\ \mathcal{O}\left(\left(\frac{n^2k}{p}\right)^{2/3} + \frac{n^2}{p^{2/3}}\right) & k/p \leq n \leq k\cdot\sqrt{p} \quad \text{(three large dimensions)} \\ \mathcal{O}\left(n^2\right) & n < k/p \quad \text{(one large dimension)} \end{cases}$$

To not be dominated by the matrix inversion, we therefore need $W_{\text{MM}}(n,k,p) > W_{\text{Rec-Tri-Inv}} \Rightarrow W_{\text{MM}}(n,k,p) > \mathcal{O}\left(\frac{n^2}{p^{2/3}}\right)$. This is only the case when $n < k$ which makes sense, since otherwise the full inversion would obviously be the dominating part, as the matrix is larger then the right hand side.

Latency cost: This approach leads to a total latency cost of

$$\begin{aligned} S_{\text{Inv-TRSM}}(n,k,p,n_0) &= S_{\text{Rec-Tri-Inv}}(n,p,n_0) + W_{\text{MM}}(n,k,p) \\ &= \mathcal{O}\left(\log^2 p\right), \end{aligned}$$

for all of the three cases.

Flop cost: The flop cost of this algorithm is

$$\begin{aligned} F_{\text{Inv-TRSM}}(n,k,p,n_0) &= F_{\text{Rec-Tri-Inv}}(n,p,n_0) + W_{\text{MM}}(n,k,p) \\ &= \mathcal{O}\left(\frac{n^3}{p}\right) + \mathcal{O}\left(\frac{n^2k}{p}\right). \end{aligned}$$

Therefore, the algorithm requires substantially more computation than Algorithm 3 when $n > k$.

5.2 Recursive TRSM with Block Inversion

In this section, we discuss an algorithm to solving TRSM recursively with a complete inversion as the base-case. We provide a cost analysis for this method as well as optimal base-case choices dependent on the relative matrix sizes.

Algorithmic approach: The goal is to keep the latency as low as possible, without increasing the asymptotic flop or bandwidth cost. We want to achieve, that the multiplication with the right hand side is the bandwidth-wise dominant part and not the the matrix inversion (which has cost $\mathcal{O}\left(\frac{n^2}{p^{2/3}}\right)$), since the former is bandwidth we necessarily have to pay. To achieve this, it is necessary that $n \leq k$. Therefore, recursive steps are taken as in Algorithm 1 in Section 3.2 until $n \leq k$. Once a recursive level is reached, where n

is sufficiently small relative to k, a switch to Algorithm 3 is performed, as the triangular matrix inversion should no longer be a bottleneck. The resulting algorithm, that is referred to as Rec-Inv-TRSM, is the same as Algorithm 1, except that the base-case is replaced by a call to Algorithm 3. It is shown as Algorithm 4.

Algorithm 4: $X = \text{Rec-Tri-Inv-TRSM}(L, B, n, k, \Pi, n_0)$

Require: *L is a lower triangular $n \times n$ matrix and B is a rectangular $n \times k$ matrix, both distributed over $|\Pi| = p$ processors.*

if $n \leq n_0$ **then**

  $X = \text{Inv-TRSM}(L, B, n, k, \Pi)$

end

else

  Subdivide L into $n/2 \times n/2$ blocks, $L = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix}$

  Subdivide B and X into $n/2 \times k$ blocks, $B = \begin{bmatrix} B_1 \\ B_2 \end{bmatrix}$, $X = \begin{bmatrix} X_1 \\ X_2 \end{bmatrix}$.

  Compute $X_1 = \text{Rec-Tri-Inv-TRSM}(L_{11}, B_1, n/2, k, \Pi, n_0)$.

  Compute $B_2' = B_2 - \text{MM}(L_{21}, X_1, n/2, k, p)$.

  Compute $X_2 = \text{Rec-Tri-Inv-TRSM}(L_{22}, B_2', n/2, k, \Pi, n_0)$.

  Assemble X from the $n/2 \times k$ blocks, $X = \begin{bmatrix} X_1 \\ X_2 \end{bmatrix}$.

end

Ensure: $LX = B$.

Case 1: If $n \leq k$ we do not need any steps of the first (recursive) TRSM approach and can directly invert the matrix. This gives the results show in Section 5.1.

Case 2: The more interesting case is where $n > k$ and we can use steps of both approaches.

Bandwidth cost: Here we show that, using inversion, we do not get a higher bandwidth than with the initial TRSM approach. The base-case costs us

$$W_{\text{Rec}-\text{Inv}-\text{TRSM}}(n_0, k, p, n_0) = \mathcal{O}\left(\frac{n_0^2}{p^{2/3}}\right)$$

bandwidth and since we have $\frac{n}{n_0}$ base-cases, that are all executed sequentially, this leads us to a total cost for all the base-cases of

$$W_{\text{base}-\text{cases}}(n, k, p, n_0) = \mathcal{O}\left(\frac{n \cdot n_0}{p^{2/3}}\right)$$

With the cost for the initial matrix multiplication as stated in Section 2.1, this leads to a total cost of

$$\begin{aligned} W_{\text{Rec-Inv-TRSM}}(n,k,p,n_0) &= W_{\text{base-cases}}(n,k,p,n_0) + W_{\text{MM}}(n,k,p) \\ &= \frac{n \cdot n_0}{p^{2/3}} + W_{\text{MM}}(n,k,p). \end{aligned}$$

Since the goal is to be dominated by the initial MM, i.e.

$$W_{\text{Rec-Inv-TRSM}}(n,k,p,n_0) = \mathcal{O}\left(W_{\text{MM}}(n,k,p)\right)$$

n_0 is chosen accordingly:
One large dimension: This case never occurs in $n > k$. We would be in Case 1 and do the direct inversion.
Two large dimensions: $n_0 = kp^{1/6}$
Three large dimensions: $n_0 = n^{1/3}k^{2/3}$
This leads to the following total bandwidth cost:

$$W_{\text{Rec-Inv-TRSM}}(n,k,p,n_0) =$$
$$= \begin{cases} \mathcal{O}\left(\left(1+\log\left(\frac{n}{k\sqrt{p}}\right)\right)\frac{nk}{\sqrt{p}}\right) & n > k\cdot\sqrt{p} \quad \text{(two large dimensions)} \\ \mathcal{O}\left(\left(\frac{n^2k}{p}\right)^{2/3}\right) & k/p \leq n \leq k\cdot\sqrt{p} \quad \text{(three large dimensions)} \\ \mathcal{O}\left(\frac{n^2k}{p}\right) & n < k/p \quad \text{(one large dimension)} \end{cases}$$

Latency: The total latency is given by the number of base-cases of the first part of the algorithm times the base-case latency (that is S_{TRI}). Therefore we have (in $\alpha-\beta-\gamma$)

$$S_{\text{Rec-Inv-TRSM}}(n,k,p,n_0) = \frac{n}{n_0}\cdot S_{\text{Rec-Tri-Inv}}$$
$$= \begin{cases} \mathcal{O}\left(\frac{n}{kp^{1/6}}\cdot\log^2 p\right) & n > k\cdot\sqrt{p} \quad \text{(two large dimensions)} \\ \mathcal{O}\left(\left(\frac{n}{k}\right)^{2/3}\cdot\log^2 p\right) & k/p \leq n \leq k\cdot\sqrt{p} \quad \text{(three large dimensions)} \\ \mathcal{O}\left(\log^2 p\right) & n < k/p \quad \text{(one large dimension)} \end{cases}$$

This is an improvement over the analysis given in Section 3.2.

Flop costs: With the given choice of n_0 and p_0, it can be shown that the flop cost is not asymptotically higher than the one for a standard implementation of TRSM. The flop cost of Rec-TRSM is

$$F_{\text{Rec-TRSM}}(n,k,p) = \mathcal{O}\left(\frac{n^2k}{p}\right),$$

whereas the cost of Algorithm 4 is

$$F_{\text{Rec}-\text{Inv}-\text{TRSM}} = \mathcal{O}\left(\frac{n^2k}{p} + \frac{n}{n_0}\frac{n_0^3}{p}\right) = \mathcal{O}\left(\frac{n^2k}{p} + \frac{nn_0^2}{p}\right).$$

Therefore it is desired that $\frac{nn_0^2}{p} < \frac{n^2k}{p}$, which implies the criterion that $n_0 < \sqrt{nk}$. It is demonstrated that the choice of n_0 made to obtain the desired bandwidth cost, also satisfies this criterion and therefore does not require additional computation work asymptotically.

One large dimension: The choice of the base-case size is $n_0 = n$ and with that

$$n_0^2 \leq n^2 < \frac{nk}{p} < nk$$
$$n_0 < \sqrt{nk}$$

Two large dimensions: The choice of the base-case size is $n_0 = kp^{1/6}$. Since two large dimensions also impose the constraint $n > k\sqrt{p}$,

$$k < \frac{n}{\sqrt{p}},$$
$$\sqrt{k} < \frac{\sqrt{n}}{p^{1/4}},$$
$$n_0 < \frac{\sqrt{k}\sqrt{n}}{p^{1/4}} \cdot p^{1/6} = \frac{\sqrt{nk}}{p^{1/12}} < \sqrt{nk}$$

Thus, this choice of n_0 guarantees that the computation cost involved in the triangular matrix inversion is always of low order when there are two large matrix dimensions at the beginning of the recursion.

Three large dimensions: The choice of the base-case size is $n_0 = n^{1/3}k^{2/3}$. Since three large dimensions also impose the constraint $n > k$, we directly see that

$$n_0 \leq n^{1/3}k^{2/3} < \left(n^{1/3}\right)\left(n^{1/6}k^{1/2}\right) \leq \sqrt{nk}$$

which implies that the computation cost of the inversion may be of leading order. Therefore, in practice, when $n = k$, it may make sense to take a few steps of recursion in Algorithm 4, before performing the inversion.

5.3 Summary

In this part all the results derived in this section are summed up and a table with the total cost of all algorithms looked at is given.

	1 Large Dimension		
	W	S	F
MM	n^2	$\log p$	$\frac{n^2k}{p}$
TRSM_{Rec}	n^2	$\log p$	$\frac{n^2k}{p}$
TRSM_{Inv}	n^2	$\log^2 p$	$\frac{n^2k}{p}$
$\text{TRSM}_{\text{RecInv}}$	n^2	$\log^2 p$	$\frac{n^2k}{p}$
	2 Large Dimensions		
	W	S	F
MM	$\frac{nk}{\sqrt{p}}$	$\log p$	$\frac{n^2k}{p}$
TRSM_{Rec}	$\left(1+\log\left(\frac{n}{k\sqrt{p}}\right)\right)\frac{nk}{\sqrt{p}}$	$\min\left(1,\frac{\sqrt{p}}{k}\right) n\log p$	$\frac{n^2k}{p}$
TRSM_{Inv}	$\frac{nk}{\sqrt{p}}+\frac{n^2}{p^{2/3}}$	$\log^2 p$	$\frac{n^2k}{p}+\frac{n^3}{p}$
$\text{TRSM}_{\text{RecInv}}$	$\left(1+\log\left(\frac{n}{k\sqrt{p}}\right)\right)\frac{nk}{\sqrt{p}}$	$\frac{n}{kp^{1/6}}\log^2 p$	$\frac{n^2k}{p}$
	3 Large Dimensions		
	W	S	F
MM	$\left(\frac{n^2k}{p}\right)^{2/3}$	$\log p$	$\frac{n^2k}{p}$
TRSM_{Rec}	$\left(\frac{n^2k}{p}\right)^{2/3}$	$\min\left(n,\left(\frac{np}{k}\right)^{2/3}\right)\log p$	$\frac{n^2k}{p}$
TRSM_{Inv}	$\left(\frac{n^2k}{p}\right)^{2/3}+\frac{nk}{\sqrt{p}}$	$\log^2 p$	$\frac{n^2k}{p}+\frac{n^3}{p}$
$\text{TRSM}_{\text{RecInv}}$	$\left(\frac{n^2k}{p}\right)^{2/3}$	$\left(\frac{n}{k}\right)^{2/3}\log^2 p$	$\frac{n^2k}{p}$

Table 5.1: Asymptotic upper bounds for bandwidth- (W), latency- (S) and flop-costs (F) for all the algorithms mentioned

In Table 9.1, the costs for all the algorithms presented in Sections 3 and 5 are shown as asymptotic upper bounds for a triangular matrix $L \in \mathbb{R}^{n\times n}$ and a right hand side $B \in \mathbb{R}^{n\times k}$ with p processors working on the task.

Chapter 6

Implementation

In this chapter, some implementation details are given.

For the implementation of the given algorithms, ScaLAPACK [1] was used as a base level library to perform the base-case calculations as well as the matrix multiplications. It is important to note that with that choice, the optimal communication costs, that were assumed throughout the theoretical part, are not achieved for the one and the three large dimensions case, since ScaLAPACK only implements a two dimensional matrix multiplication. For a efficient work distribution while staying on a relatively simple distribution model, a block cyclic distribution of the matrix was chosen as shown in Figure 6.1. Each processor owns the parts of the matrix that are colored in its color and with that as soon as we multiply the four-by-four blocks, we have all the processors working on that and not only on the top level matrix multiplication. With this choice we are able to ensure that after some blocked steps, all processors are working on the matrix multiplications. This was especially important as we showed that the top level matrix multiplications are always the most expensive. For efficiency reasons, all algorithms were implemented in an iterative scheme.

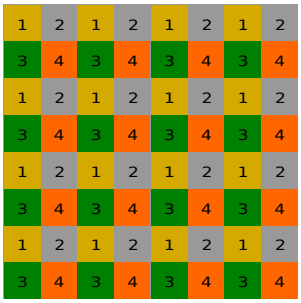

Figure 6.1: Representation of a block cyclic distribution of the matrix

Inversion: For the inversion of the base-cases, the algorithm proposed in section 5.1 was implemented. To ensure that the base-case of the inversion is done sequentially, the block size for the distribution was chosen to be the base-case size of the inversion.

In each "level" the number of matrix multiplications decreases by two as the size of the matrices increases by two until we reach the top level, as can be seen in Figure 6.2: In the first level (denoted by 1), the four green squares are multiplied. In the second level, the two bigger, red matrices are multiplied and in the final step, the blue square matrix is multiplied leading to a complete inversion.

Triangular Solver: The triangular solve itself was using this inversion within each base-case and then did the required updates with ScaLAPACK's triangular matrix multiplication and dense matrix multiplication respectively, as can be seen in Algorithm 5. The updates on the right hand side done in update_B are always done up to the part that was calculated, as it was proposed in the recursive part in Section 5.2. These updates are denoted as the consecutive numbers in Figure 6.2: After the first base-case has been used to calculate X_1, the small (green) update 1 is done on B_2. After solving for X_2 in the second step, the update of B affects B_3 and B_4 as the bigger (red) block 2 is updated. This goes on until the last base-case is handled.

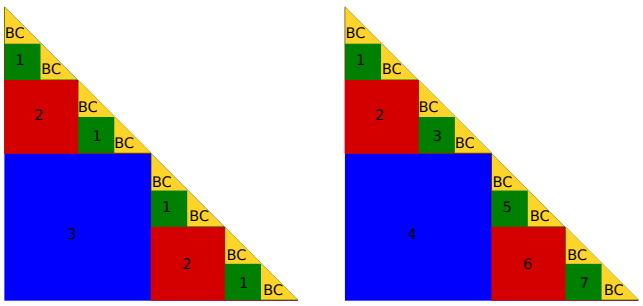

Figure 6.2: Recursion in the triangular matrices: Inversion steps (left), triangular solve (right)

Algorithm 5: Rec-Inv-TRSM(L, B)

for *i in number_of_basecases* **do**
  Rec-Inv($L_{i,i}$)

for *i in number_of_basecases* **do**
  X_i = ScaLAPACK_mm($L_{i,i}^{-1}$, B_i)
  update_B(X_i, L)

Chapter 7

Experimental Results

In this section, results of the performed experiments are provided. After a description of the experimental setup, including information about the machine used, the performance of the proposed algorithms is evaluated.

Experimental setup: The experiments were performed on a Cray XC40 with 1256 nodes. Each node contains two Intel Xenon E5-2690 v3 CPUs with 12 cores each and 64 GB of memory. The nodes are connected by the "Aries" proprietary interconnect from Cray, with a dragonfly network topology.

Compilers and libraries: LAPACK and ScaLAPACK [1] was used from the provided precompiled cray-libsci library version 13.0.4. For the matrix storage and correctness checks, the Eigen [14] library version 3.2.5 was used. All programs were compiled with gcc 4.9.2 with the optimization flag O3.

Initial Conditions: The matrices were created locally using the drand48 random number generator within the range $[1, 101)$, where L is lower triangular and B is dense.

Restrictions: To focus on the algorithmic improvements, the timing was started after the MPI-communicators were initialized, the Cblas grid was created and the local matrices were initialized. Each simulation was ran six to eleven times and the first run was neglected. To summarize the data, as suggested in [15, 16], the harmonic mean

$$\bar{x}^{(h)} = \frac{n}{\sum_{i=1}^{n} (1/x_i)}$$

was used instead of the arithmetic mean. Also for each processor number, the two-sided 95%-confidence intervals of each set of experimental results is shown. These intervals were calculated as mentioned in [16] based on Students t distribution:

$$CI : \left[\bar{x} - t(n-1, \alpha/2)s/\sqrt{n}, \bar{x} + t(n-1, \alpha/2)s/\sqrt{n}\right]$$

where s denotes the sample standard deviation

$$s = \sqrt{\left(\sum_{i=1}^{n}(x_i - \bar{x})^2\right) / (n-1)}$$

In each of the following graphs, the mean value is plotted with a larger icon and lines, all the single runs are plotted with the smaller icons and the 95% confidence interval is plotted in black.

Base-Case Size: We expected that in real simulations, the theoretically optimal base-case size can be the non-optimal one for some problems. This is due to constants of the asymptotic complexities as well as the two dimensional implementation of the matrix matrix multiplication that we used. Therefore ranges around the optimal base-case size were chosen for each experiment ($n_{\text{opt}}/4, n_{\text{opt}}/2, n_{\text{opt}}, 2 \cdot n_{\text{opt}}, 4 \cdot n_{\text{opt}}$). Only the best performing base-case size was accounted for.

Strong scaling: The strong scaling plots were produced to show the average performance over the runs. Therefore the total flop count ($n^2 \cdot k$ for TRSM and $\frac{1}{3}n^3 + \frac{2}{3}n$) was divided by the execution time which led to the flop-count per second:

$$\mathrm{G}_{\text{TRSM}}(t_{\text{exec}}(p), n, k) = \frac{n^2 k}{t_{\text{exec}}(p) \cdot 10^{-9}}$$
$$\mathrm{G}_{\text{Inv}}(t_{\text{exec}}(p), n) = \frac{\frac{1}{3} \cdot n^3 + \frac{2}{3}n}{t_{\text{exec}}(p) \cdot 10^{-9}}$$

For a perfectly parallelizeable algorithm with no parallelization overhead cost, this property should scale like p, since the execution time in this case would scale this way.

Weak scaling: For the weak scaling plots, the property G was reused and divided by the machine peak performance for each set of nodes used. Therefore the plotted property is:

$$\mathrm{P}(t_{\text{exec}}(p), n, k, p) = \frac{\mathrm{G}(t_{\text{exec}}(p), n, k)}{p \cdot \text{Peak\_per\_processor}}$$

For the peak performance 41.6 GF / core has been assumed. Each processor runs on one core only. For a perfectly parellelizeable algorithm with no overhead cost, this property should be constant at 1, since we would run at peak performance with every processor for the whole time.

Inversion: One of the benchmarks we did, was to benchmark our implementation for the inversion against the inversion ScaLAPACK provides. The

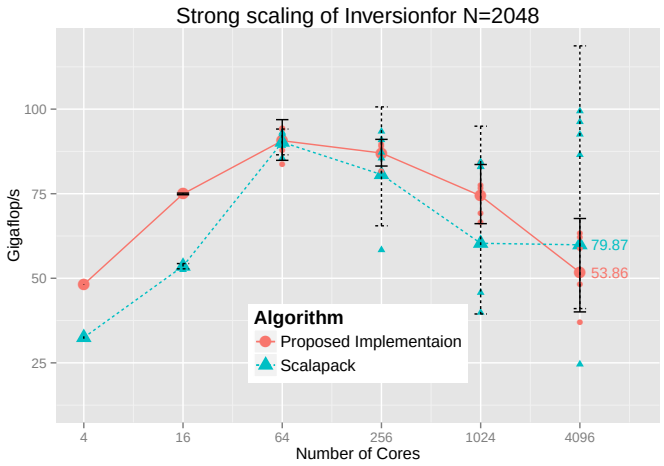

Figure 7.1: Strong scaling of the inversion with N=2048

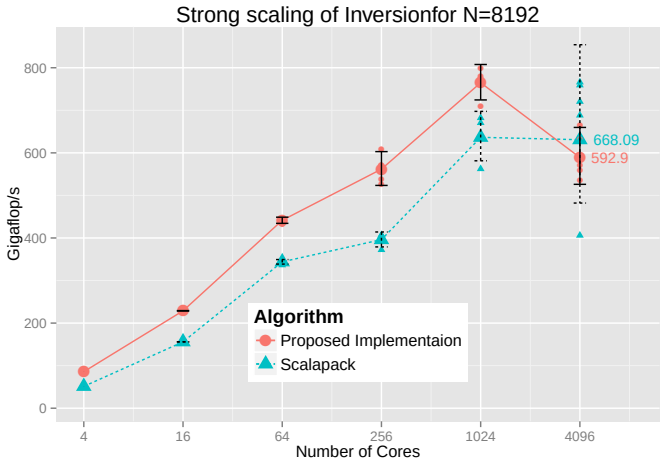

Figure 7.2: Strong scaling of the inversion with N=8192

results can be seen in Figures 7.1,7.2 and 7.3. One can see, that this part of the TRSM-solver was able to to beat ScaLAPACK in strong as well as in weak scaling. It is interesting to observe, that for smaller problems (up to $N = 8192$), the problems appear to be too small to get the full benefit of 4096 ranks working on it. But the important thing to note is, that for problems large enough, the proposed algorithm is a real improvement compared to ScaLAPACK, as one gets more than a factor of 1.6 in Gigaflop per second for $N = 32768$. The weak scaling has been started at $N = 1024$ and $p = 4$ and increased N by 2 as p increased by 4 to again keep memory usage per processor constant. In the weak scaling plot, visible in Figure 7.4, it can be

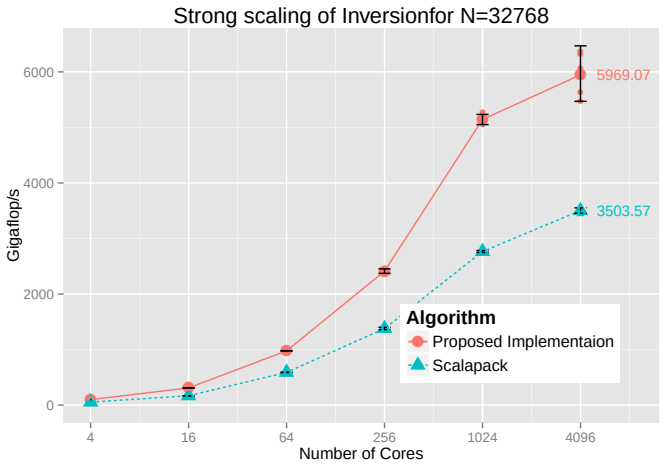

Figure 7.3: Strong scaling of the inversion with N=32768

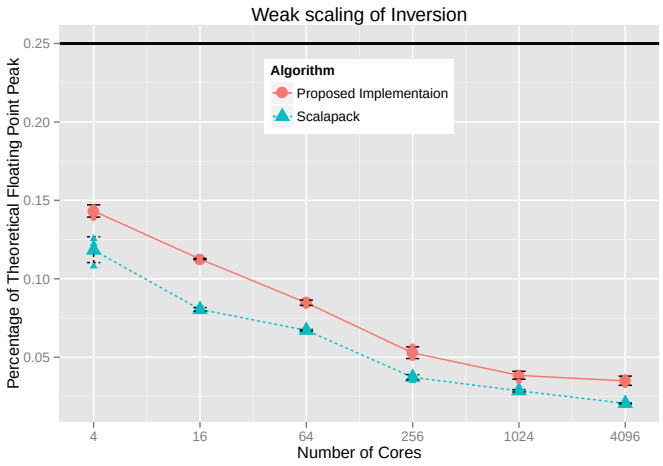

Figure 7.4: Weak scaling starting with N=1024 for p=4

seen, that the proposed method is strictly better in terms of percentage of peak performance.

Three Large Dimensions: Benchmarks for the algorithm were performed for the three large dimensions case. The strong scaling plots for different matrix sizes, where $L \in \mathbb{R}^{N \times N}$ and $B \in \mathbb{R}^{N \times K}$, are provided for the presented implementation as well as ScaLAPACKs algorithm in Figures 7.5,7.6 and 7.7. It is clearly visible that the scaling is very poor for small matrix sizes, but as we increase the sizes, the scaling becomes very good. Unfortunately, the overall performance is still worse than what ScaLAPACK offers.

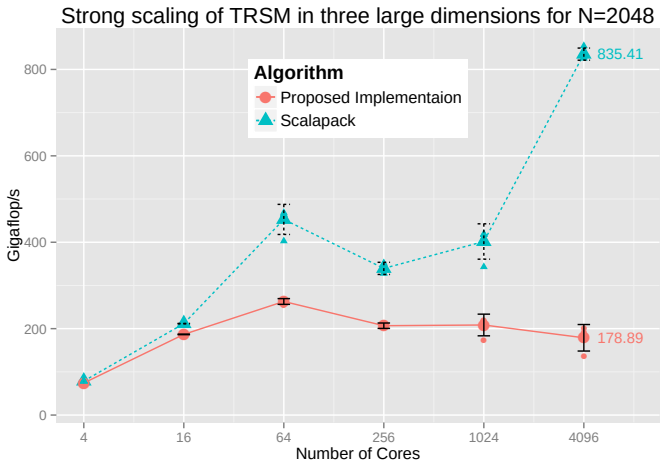

Figure 7.5: Strong scaling for N=K=2048

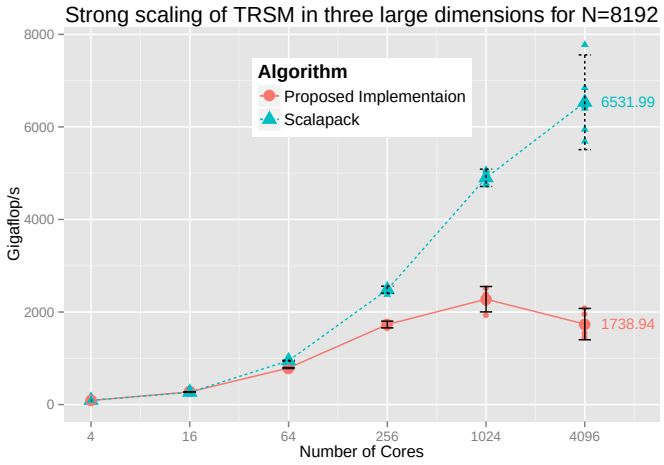

Figure 7.6: Strong scaling for N=K=8192

After profiling the runs, it was easy to see that the biggest problem does not lie in the inversion part but in the very slow triangular matrix - dense matrix multiplication (TRMM). It is even slower than doing the complete triangular solve (TRSM), even though the maximal amount of parallelism possible for TRSM is only nk, whereas it is n^2k for the TRMM. The weak scaling plot can be seen in Figure 7.8. As for the weak scaling, it was observed that the scaling was not as promising as the strong scaling suggests. The results were created with the following set of parameters: Starting with $p = 4$ and $N = K = 1024$, N and K were both increased by a factor of two as p increased by a factor of four.

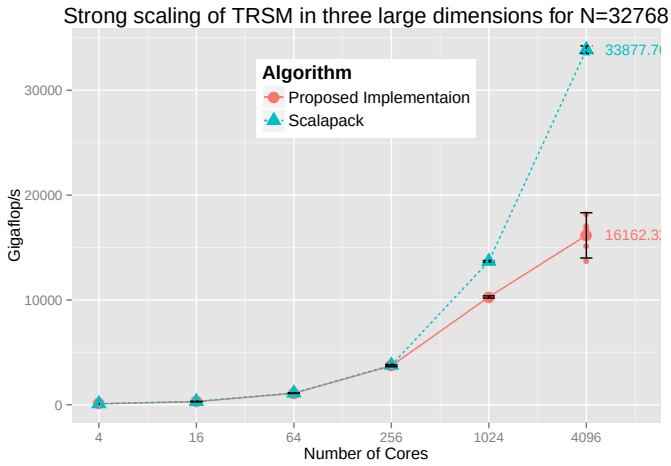

Figure 7.7: Strong scaling for N=K=32768

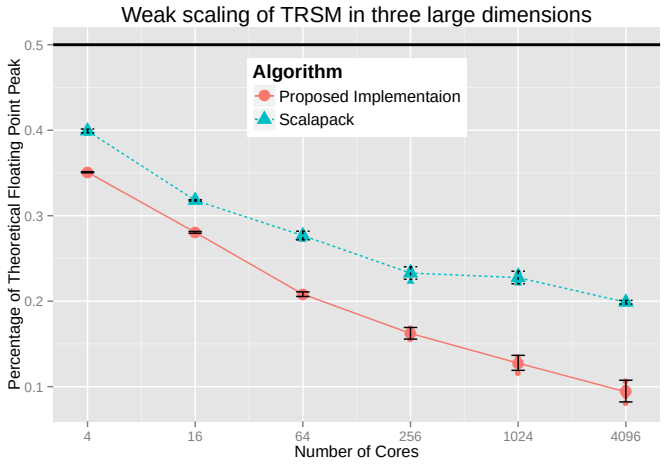

Figure 7.8: Weak scaling starting with N=1024 for p=4

Two Large Dimensions: Since the TRMM was such a dominating factor in the previous case, one could hope for better performance as the right hand side size decreases. The strong scaling plots were created for several sizes n of the matrix $L \in \mathbb{R}^{n \times n}$ and kept the width size of the right hand side $K \in \mathbb{R}^{n \times k}$ constant. The observed results can be seen in Figures 7.9,7.10 and 7.11. One can see that for smaller numbers of processors, the discussed approach works very well. Unfortunately, the performance relative to ScaLAPACK decreases at the very end due to a spike in ScaLAPACKs performance, which is unexplainable for now. But the interesting part is, that as we decrease the time spent in the TRMM, the more important the actual new

content becomes and this shows to be efficient. The weak scaling has been started at $N = 1024$ and $p = 4$ and increased N by two as p increases by four to again keep memory usage per processor constant. The results are shown in Figure 7.12.

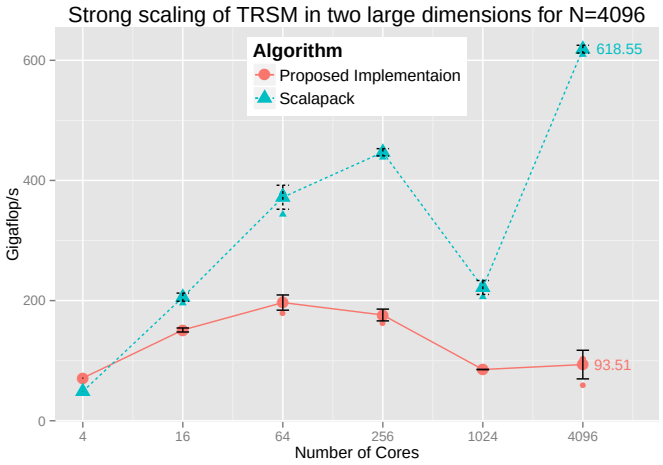

Figure 7.9: Strong scaling for N=4096 with K=512

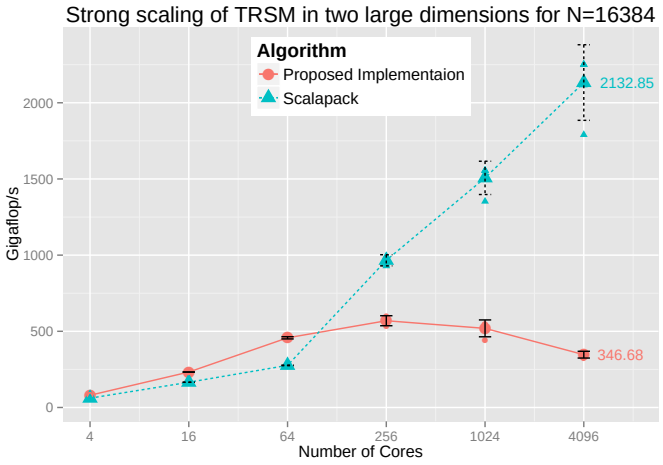

Figure 7.10: Strong scaling for N=16384 with K=512

One Large Dimension: Since we already found out that the inversion is fast and the TRMM is slow, we decided to not do any benchmarks for the one large dimensions case to save computing time, as nothing interesting could have been observed there.

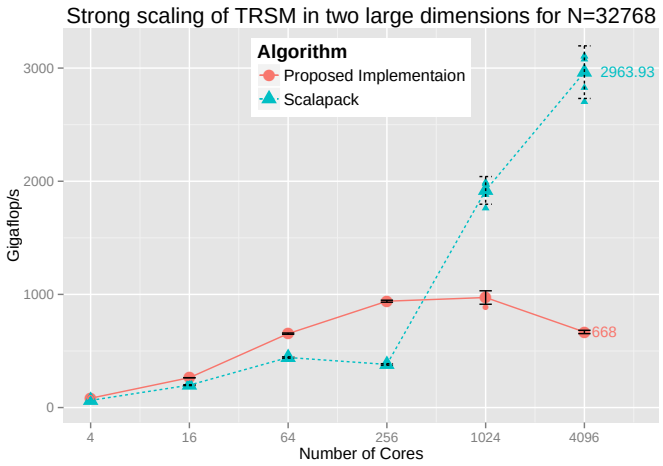

Figure 7.11: Strong scaling for N=32768 with K=512

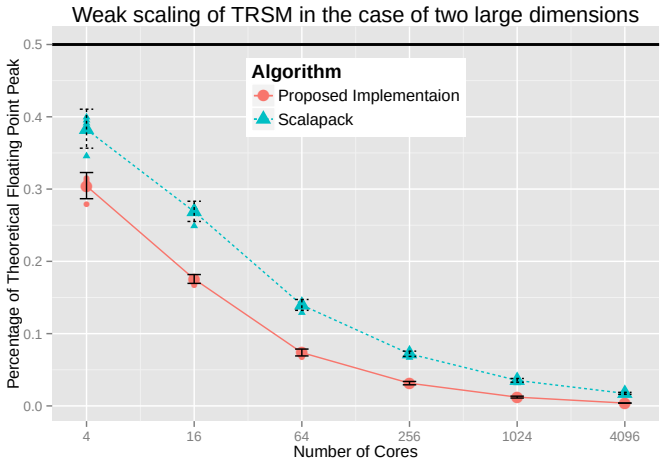

Figure 7.12: Weak scaling starting with N=1024 for p=4

Summary of the results: The proposed algorithm does not bring the expected benefits due to the slow implementation of the triangular matrix multiplication. With less time spent in the triangular matrix multiplication, better performance for the proposed algorithm can be obtained. Nevertheless, the main part of the algorithm, being the faster inversion of triangular blocks, shows a good increase of performance over the routine implemented in ScaLAPACK [1].

Chapter 8

Further Work

In this chapter, further work relevant to this topic will be presented.

Stability Analysis: As mentioned in Section 2, a stability analysis of the proposed method for blocked inversion could be done. Such an approach is partially described in [2]. For the problems looked at in this work, where every lower triangular matrix was good conditioned and therefore no problems have risen in correctness compared to the results a regular, iterative scheme provided.

Optimizing Triangular Matrix Multiplications: The results show very clearly that, in order to make the proposed method work, a much faster triangular matrix multiplication is needed. Due to the limited area of application compared to a dense matrix multiplication or even a triangular solve, the optimization in ScaLAPACK is presumably rather low. Therefore, most likely, new code has to be developed at this point. Another limitation of the current state is that we always use a two-dimensional grid. To get the asymptotic optimal cost, this should be extended to at least cover three-dimensional as well as one-dimensional grid layouts.

Adapting the structure for arbitrary matrix sizes: So far, only problems with matrix sizes chosen as powers of two are looked at. This made the recursive algorithms more simple to implement, as there are no leftovers. An improvement to the code could be made, such that general sizes are usable. The most common approach of doing so, would be to extend the matrix to fit a power of two with a block of the identity matrix at the bottom and add zero rows to the right hand side.

Further optimization: The results show, that the code has a performance gap that exists as subroutines used show a significant lack of scaling. Also one can see that the performance ends up never being higher than 30 % of the peak performance for a large number of processors, and for very well

optimized code, one can expect to get a factor of about two more in peak performance.

Chapter 9

Conclusion

This work presented a new approach to a communication avoiding way of solving a triangular system for multiple right hand sides (TRSM). TRSM serves as a subroutine for a lot of algorithms in dense linear algebra like the LU decomposition and it is the critical assert for solving linear systems of equations. It has been shown that the algorithm we propose asymptotically uses the same bandwidth and flop costs as the standard iterative scheme for solving the TRSM problem, but decreases the latency by a factor of $\frac{p^{2/3}}{\log p}$ for the case of three large dimensions as well as for the case of two large dimensions. To achieve the decreased latency one has to carefully pick the base-case size to not be dominated by the bandwidth or the flop costs. For all three cases of relative matrix sizes we presented base-case sizes that are optimal. For the case of one large dimension, the algorithm does not perform any better than the existing one due to the fact that single processor work on the left side is always preferred and therefore no gain was achieved. The summarized costs are shown in Table 9.1. Due to the fact that only asymptotic upper bounds are presented for the matrix multiplications, we were only able to give asymptotic upper bounds for the performance of our algorithm. With this decrease of latency our algorithm is very promising for solving linear systems faster on machines with a large number of processors. This is especially true as for larger systems, the communication cost is more of a bottleneck than it is on small machines. Also this work opens up the question if one should go back and consider the new triangular inversion as a subroutine for other applications as it was done by Tiskin for the LU factorization in [7].

Experiments with a not heavily optimized version of this algorithm were performed. We used ScaLAPACKs two dimensional matrix multiplication for the calculations of our algorithm. The results showcased that the new approach of doing the inversion turned out to bring a notable speedup, whereas, due to the lack of a well optimized triangular matrix multiplica-

	1 Large Dimension		
	W	S	F
TRSM_{Rec}	n^2	$\log p$	$\frac{n^2k}{p}$
$\text{TRSM}_{\text{RecInv}}$	n^2	$\log^2 p$	$\frac{n^2k}{p}$
	2 Large Dimensions		
	W	S	F
TRSM_{Rec}	$\left(1+\log\left(\frac{n}{k\sqrt{p}}\right)\right)\frac{nk}{\sqrt{p}}$	$\min\left(1, \frac{\sqrt{p}}{k}\right) n \log p$	$\frac{n^2k}{p}$
$\text{TRSM}_{\text{RecInv}}$	$\left(1+\log\left(\frac{n}{k\sqrt{p}}\right)\right)\frac{nk}{\sqrt{p}}$	$\frac{n}{kp^{1/6}}\log^2 p$	$\frac{n^2k}{p}$
	3 Large Dimensions		
	W	S	F
TRSM_{Rec}	$\left(\frac{n^2k}{p}\right)^{2/3}$	$\min\left(n, \left(\frac{np}{k}\right)^{2/3}\right)\log p$	$\frac{n^2k}{p}$
$\text{TRSM}_{\text{RecInv}}$	$\left(\frac{n^2k}{p}\right)^{2/3}$	$\left(\frac{n}{k}\right)^{2/3}\log^2 p$	$\frac{n^2k}{p}$

Table 9.1: Summarized upper bounds for bandwidth- (W), latency- (S) and flop-costs (F) for all the algorithms mentioned

tion, the time to solution for the algorithm presented to solve triangular systems still turns out to be higher than the reference. We were able to see that the problem sizes for which we did our experiments were rather small and with that, the percentage of peak performance was lower then expected. But the trends that the graphs show are promising.

Bibliography

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

[2] Erik Elmroth, Fred Gustavson, Isak Jonsson, and Bo Kågström, "Recursive blocked algorithms and hybrid data structures for dense matrix library software," *SIAM review*, vol. 46, no. 1, pp. 3–45, 2004.

[3] Edgar Solomonik and James Demmel, "Communication-optimal parallel 2.5 D matrix multiplication and LU factorization algorithms," in *Euro-Par 2011 Parallel Processing*, pp. 90–109. Springer, 2011.

[4] Fred G Gustavson, "Recursion leads to automatic variable blocking for dense linear-algebra algorithms," *IBM Journal of Research and Development*, vol. 41, no. 6, pp. 737–755, 1997.

[5] J. Demmel, D. Eliahu, A. Fox, S. Kamil, B. Lipshitz, O. Schwartz, and O. Spillinger, "Communication-optimal parallel recursive rectangular matrix multiplication," in *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, May 2013, pp. 261–272.

[6] Alok Aggarwal, Ashok K Chandra, and Marc Snir, "Communication complexity of prams," *Theoretical Computer Science*, vol. 71, no. 1, pp. 3–28, 1990.

[7] Alexander Tiskin, "Bulk-synchronous parallel gaussian elimination," *Journal of Mathematical Sciences*, vol. 108, no. 6, pp. 977–991, 2002.

[8] Michael T Heath and Charles H Romine, "Parallel solution of triangular systems on distributed-memory multiprocessors," *SIAM Journal on Scientific and Statistical Computing*, vol. 9, no. 3, pp. 558–588, 1988.

[9] Edgar Solomonik, Erin Carson, Nicholas Knight, and James Demmel, "Tradeoffs between synchronization, communication, and computation in parallel linear algebra computations," in *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, New York, NY, USA, 2014, SPAA '14, pp. 307–318, ACM.

[10] Grey Ballard, James Demmel, Benjamin Lipshitz, Oded Schwartz, and Sivan Toledo, "Communication efficient gaussian elimination with partial pivoting using a shape morphing data layout," in *Proceedings of the twenty-fifth annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 2013, pp. 232–240.

[11] Benjamin Lipshitz, "Communication-avoiding parallel recursive algorithms for matrix multiplication," Tech. Rep., EECS Department, University of California, Berkeley, 2013.

[12] Jeremy J Du Croz and Nicholas J Higham, "Stability of methods for matrix inversion," *IMA Journal of Numerical Analysis*, vol. 12, no. 1, pp. 1–19, 1992.

[13] Raphael Hunger, *Floating point operations in matrix-vector calculus*, Munich University of Technology, Inst. for Circuit Theory and Signal Processing Munich, 2005.

[14] Gaël Guennebaud, Benoît Jacob, et al., "Eigen v3," http://eigen.tuxfamily.org, 2010.

[15] Philip J Fleming and John J Wallace, "How not to lie with statistics: the correct way to summarize benchmark results," *Communications of the ACM*, vol. 29, no. 3, pp. 218–221, 1986.

[16] T. Hoefler and R. Belli, "Scientific Benchmarking of Parallel Computing Systems," Nov. 2015, Accepted at IEEE/ACM International Conference on High Performance Computing, Networking, Storage and Analysis (SC15).