JOHANNES KEPLER
UNIVERSITY LINZ

Submitted by
Stefan Amberger

Submitted at
Research Institute for Symbolic Computation

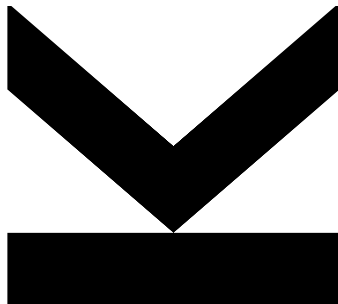
Supervisor
**Univ.-Prof. Dr.
Peter Paule**

Supervisor
**Dr.sc.techn. Dipl.-Ing.
Volker Strumpen**

Supervisor
**A.Univ.-Prof. DI Dr.
Wolfgang Schreiner**

Nov 2018

A Parallel, In-Place, Rectangular Matrix Transpose Algorithm



Master Thesis
to obtain the academic degree of
Diplom-Ingenieur
in the Master's Program
Computermathematics

**JOHANNES KEPLER
UNIVERSITY LINZ**
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

Statement of Originality

I hereby declare, that the work contained in this thesis has not been previously submitted for a degree or diploma at any higher education institution.

To the best of my knowledge and belief, the thesis contains no material previously published or written by another person except where due references are made.

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe. Die vorliegende Masterarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Signature

Date

Abstract

This thesis presents a novel algorithm for **T**ransposing **R**ectangular matrices **I**n-place and in **P**arallel (TRIP) including a proof of correctness and an analysis of work, span and parallelism.

After almost 60 years since its introduction, the problem of in-place rectangular matrix transposition still does not have a satisfying solution. Increased concurrency in todays computers, and the need for low-overhead algorithms to solve memory-intense challenges are motivating the development of algorithms like TRIP.

The algorithm is based on recursive splitting of the matrix into sub-matrices, independent, parallel transposition of these sub-matrices, and subsequent combining of the results by a parallel, perfect shuffle.

We prove correctness of the algorithm for different matrix shapes (ratios of dimensions), and analyze work and span: For an $M \times N$ matrix, where $M > N$, and both M and N are powers of two, TRIP has work

$$W_1^{\texttt{TRIP}}(M, N) = \Theta\left(MN \log \frac{M}{N} \log N\right)$$

span

$$W_\infty^{\texttt{TRIP}}(M, N) = \Theta\left(\log \frac{M}{N} \log^2 N + \log^2 \frac{M}{N} \log N\right)$$

and a parallelism of

$$\Theta\left(\frac{MN}{\log M/N + \log N}\right)$$

Due to symmetry, the work and span of a $M \times N$ matrix with $M < N$ can be calculated by interchanging M and N in the formulas above. Compared to out-of-place algorithms, which have work $M \cdot N$, TRIP, implemented in Cilk, trades work-efficiency for parallelism and for being in-place.

Keywords: in-place, rectangular matrix transposition, parallel algorithm, Cilk

Contents

Contents III

1 Introduction 1

2 Related Work 3
- 2.1 Review of In-place Matrix Transpose Algorithms . . . 3
- 2.2 Further Influences . . . 4

3 Description of Transpose Algorithm 5
- 3.1 Basic Definitions . . . 5
- 3.2 Transpose . . . 7
- 3.3 Merge . . . 9
- 3.4 Split . . . 11
- 3.5 Square Transpose . . . 13
- 3.6 Iterators . . . 15

4 Proof of Correctness 17
- 4.1 Correctness Proof of Combine Method `merge` . . . 17
- 4.2 Correctness Proof of Combine Method `split` . . . 20
- 4.3 Correctness Proof of `TRIP` . . . 22

5 Complexity Analysis 24
- 5.1 Work . . . 24
- 5.2 Span . . . 30
- 5.3 Parallelism . . . 35
- 5.4 Interpretation of Results . . . 36
  - 5.4.1 Work and Span . . . 36
  - 5.4.2 Parallelism . . . 36
  - 5.4.3 Analysis for Matrix Configurations that are not Powers of Two . . . 37
  - 5.4.4 Optimizations to the Algorithm . . . 37

6 Experimental Results 42
- 6.1 Performance . . . 42
  - 6.1.1 Changing Aspect Ratio, Constant Matrix Size . . . 42

6.1.2 Changing Matrix Size, Constant Aspect Ratio 42
6.2 Scalability 43

7 Conclusions **49**

References **50**

1 Introduction

Transposing matrices is one of the most basic operations in linear algebra, and widely used across many fields. Notable fields of application are, among others, Fast Fourier Transform algorithms like FFTW [8] and fluid dynamics software like OpenFOAM [13], which supports rectangular transpositions, but currently does not feature an in-place algorithm. While Intels© MKL® supports in-place transpositions [10], neither BLAS nor OpenBLAS do so at the moment.

Ever since maximum clock frequencies of microprocessors reached their limit in 2005, manufacturers put multiple cores on microchips to increase their capabilities. This capability can be harnessed by parallel algorithms. The parallelism of an algorithm, the ratio of work W_1 and span W_∞, indicates on how many cores the work of the algorithm can be distributed (see [5, p. 779ff]). Except for embarrassingly parallel problems, most parallel algorithms face a work penalty; they have additional work to do, in comparison to their sequential versions. This penalty should be kept to a minimum but can be compensated to a certain degree by additional cores. The span of a parallel algorithm, however, cannot be compensated by additional cores.

Another challenge exists across devices, and drives the need for in-place algorithms: the problem-sizes that can be tackled are often limited by memory. For this reason, in-place algorithms that do not need abundant memory can be vital in solving bigger problems. The algorithm presented in this thesis does not need to allocate additional memory to store parts of the matrix being transposed. This gives a clear advantage in the case of large fluid-dynamics simulations, large data-sets in earth observation, or small amounts of available memory in embedded devices or for signal processing.

The transposition of a rectangular matrix is very simple—in principle:

```
double A[M][N], B[N][M];
for (i=0; i<M; i++)
  for (j=0; j<N; j++)
    B[j][i] = A[i][j];
```

Moving from the principle to an algorithm and from there to an in-place algorithm exposes challenges, which we tackle via a divide-and-conquer approach.

Implemented on a computer, matrices differ from their pure, mathematical counterparts. Memory is 1-dimensional, as opposed to matrices in mathematics, which are 2-dimensional. An array can be interpreted as a matrix, if sequential parts of the array are considered as the concatenation of rows (row major, as in C) or columns (column major, as in Fortran) of a matrix. In this context a transpose algorithm, as opposed to the mathematical transpose operator would look like this (applying a row-major format):

```
double A[M·N], B[N·M];
for (i=0; i<M; i++)
```

```
    for (j = 0; j < N; j++)
      B[j · M + i] = A[i · N + j];
```

Transposing a matrix ***in-place*** means to modify it with an algorithm which memory requirement is $O(1)$, and to reinterpret the result as the transpose of the original matrix, keeping the location of the object in the same memory space [8], rather than copying the content of one matrix to a second matrix in a specific pattern, as we did previously. The memory constraint of a parallel algorithm utilizing p processors is loosened to $O(p)$.

For a square matrix of dimension M an in-place algorithm is:

```
double A[M · M];
for (i = 0; i < M; i++)
  for (j = 0; j < i; j++)
    tmp = A[j · M + i];
    A[j · M + i] = A[i · M + j];
    A[i · M + j] = tmp;
```

A temporary variable is used to swap $A[j \cdot M + i]$ and $A[i \cdot M + j]$, for all i from 1 to M and j from 1 to i. As a whole, the square matrix in-place algorithm is a permutation on A, composed of $M(M+1)/2$ cycles of length 1.

In fact, all in-place matrix transpose algorithms that transpose a $M \times N$ matrix (assuming row-major order), one way or another, have to do work that is equivalent to applying the permutation presented in [4] onto each array index $x \in \{0, MN - 1\}$.

$$\pi(x) = \begin{cases} Mx \bmod MN - 1 & \text{if } x \neq MN - 1 \\ MN - 1 & \text{if } x = MN - 1 \end{cases}$$

Permutations can be written as cycles, each of which is inherently serial, if tackled directly as such. Parallelism depends upon the number of cycles of the permutation [7], which itself is dependent on the matrix dimensions. Many algorithms exist for specific or general matrix sizes, sharing the same approach, and exploiting various properties of this class of permutations [9, 17, 11, 14]. In this thesis we introduce a new approach: a recursive divide-and-conquer method, called **T**ranspose of **R**ectangular matrices, **In**-place and in **P**arallel (TRIP), that does *not* directly implement this permutation.

This algorithm has been developed by Prof. Volker Strumpen as part of his research as head of the Institute for Computer Architecture at Johannes Kepler University in Linz. The implementation, mathematical formalization, proofs of correctness, theoretical complexity analysis and benchmarks have been created by the author, under supervision of Prof. Strumpen, and reviewed by Prof. Wolfgang Schreiner and Prof. Peter Paule. Regular meetings and discussions shaped the outcome of this work.

The remainder of this thesis is organized as follows. Chapter 2 reviews the field of rectangular matrix transpose algorithms. Chapter 3 presents our parallel in-place rectangular matrix transpose algorithm TRIP. Chapter 4 contains a correctness proof of the TRIP algorithm. Chapter 5 provides a work and span analysis for matrices of dimensions that are powers of two, and an interpretation of the results. Finally, Chapter 7 concludes the thesis by summarizing the results and hinting at future work.

2 Related Work

To put our work into the perspective of those working on the same topic before us, the first part of this chapter contains known approaches to matrix transposition algorithms, their optimizations and historic development. The second part of this chapter contains work that this thesis is based on, that does not directly concern matrices and their transposes. This includes work on out-of-place matrix transpose algorithms, perfect-shuffle algorithms, and the programming language Cilk.

2.1 Review of In-place Matrix Transpose Algorithms

There are several interpretations of the problem of in-place matrix transpositions. Historically it used to be treated as a permutation problem. Recently divide and combine aspects are uncovered and used to introduce a higher and more predictable degree of parallelism.

Permutation interpretation Between 1959, when Windley introduced the problem of in-place matrix transposition [19] until 1999, all presented algorithms interpreted the in-place matrix transpose strictly as a permutation problem. In 1968 Boothroyd presented Algorithm 302 [2], Laflin and Brebner improved on this result in 1970 [12]. In 1973 Brenner separated the search for independent permutation cycles and moving the data [3], again decreasing the work of the algorithm. In 1976 Cate and Twigg derived number theoretical results to faster find independent permutation cycles [4].

One example for an in-place rectangular matrix transposition algorithm that is inspired by viewing this operation as a permutation problem is the work done by Gustavson et al. in [9]. In this paper Gustavson et al. present a work-efficient transpose algorithm called MIPT, with complexity $O(MN \log MN)$. MIPS takes configurable extra storage for efficiency. It uses the "Burn At Both Ends" programming technique to more quickly find the cycles of the permutation that constitutes the transpose. Gustavson et al. mention that one drawback of rectangular in-place transpose algorithms that directly follow permutations—as opposed to other approaches, or out-of-place algorithms—is the amount of cache misses they incur: For most matrices, permutation cycles follow a seemingly random pattern. Hence, each element access is almost guaranteed to be a cache miss.

Divide and conquer interpretation Starting in 1999, when Portnoff introduced a parallel in-place matrix transposition algorithm in [14], rectangular matrix transpose algorithms were seen as more than pure permutation problems. Portnoff applied the divide-and-conquer strategy to the problem.

In "An Efficient Parallel-Processing Method for Transposing Large Matrices in Place", he presents a matrix transpose algorithm in four steps: Step I divides the matrix into smaller element-pairs, and sub-

matrices. Step II transposes each of those sub-matrices in parallel, by applying the permutation that is specific to the sub-matrix. Step III transposes the "matrix of sub-matrices", and Step IV re-arranges the element-pairs in order to yield the transpose of the original matrix.
This algorithm applies the divide-and-conquer paradigm on the first level, to create a sets of problems that can be solved in parallel; then it calculates and applies permutations to solve those smaller problems.

2.2 Further Influences

Three main topics influenced this algorithm: parallel algorithms, shuffle algorithms, and the programming language extension Cilk. Scalable parallel algorithms are of recursive nature, using divide and combine steps along the way. These divide and combine steps are modified perfect shuffle algorithms.

The publications that influenced this thesis the most in this regard are written by Ellis et al. [6] in 2000, who presents an in-situ stable merge algorithm, a modified version of which is the base of the algorithm `merge` in this thesis, and by Jain [11] in 2004.

These parallel, recursive algorithms can relatively easily be implemented in Cilk. Cilk is an extension to the C and C++ programming languages, that allows data and task parallelism, using only three additional keywords: `cilk`, `spawn`, and `sync`. It has been developed in the 1990s at the Massachusetts Institute of Technology (MIT), and commercialized by Charles E. Leiserson et al. with the formation of Cilk Arts, Inc. In 2009 Intel Corporation acquired Cilk Arts, and included parts of Cilk into the Intel C++ compiler.
The principle behind parallel development in Cilk is that the programmer exposes parallelism in the form of functions, which the Cilk scheduler can—but doesn't have to—schedule in parallel. Leiserson et al. developed a provably efficient work-stealing scheduler [15], which allows the same program to run on single-core as well as multi-core architectures, and make use of all available resources.

At the time of finalizing this thesis, Cilk Plus has been marked as deprecated in GCC, and will be marked deprecated in the Intel Software Development Tools. Nevertheless the algorithm and programming paradigm stays alive in the form of a Tapir [16], see `http://cilk.mit.edu`. In addition to that other languages like Go implement work-stealing schedulers [18] based on Blumofe and Leisersons work [15], allowing the same programming pattern.

Appended to this thesis is the source code of TRIP in Cilk. It is compilable e.g. with gcc-7.3 using the compiler flags -fcilkplus and -lcilkrts.
The struct `ws` (short for *work* and *span*) is used to count work and span of TRIP calls, and can be removed without changing the result.

3 Description of Transpose Algorithm

In this chapter the core algorithms will be presented in the form of recursions and pseudo-code. It consists of five parts: Basic Definitions (Section 3.1, the outermost recursion, `TRIP` (Section 3.2), which calls all subroutines, the main subroutines of `merge` (Section 3.3) and `split` (Section 3.4), and the base case of the transpose recursion (Section 3.5). All algorithms are parallel, and in-place. Finally Section 3.6 covers implementation details.

3.1 Basic Definitions

In this section we define the basic notions and notation that will be used throughout Chapter 3. We define what we understand as a matrix and its transpose, what we call sub-matrices and composed matrices, and lastly we define the array-representation of a matrix, and it's inverse, the matrix-interpretation of an array.

We define a matrix and it's transpose as follows.

Definition 1 (Matrix)**.** Let $I_n := \{0, \dots, n-1\}$.

A $n \times m$ ***matrix*** A with coefficients in a ring $\mathcal{R}$ is a function $a : I_n \times I_m \longrightarrow \mathcal{R}$. We define $a_{i,j} := a(i,j)$. In other words, A is a rectangular array.

Definition 2 (Transpose)**.** Let A be an $m \times n$ matrix as defined above.

Then $B := A^T$, the ***transpose*** of A, is the function $b : I_m \times I_n \longrightarrow \mathcal{R}$ with $b_{i,j} = a_{j,i}$.

Recursive calls (3.5a and 3.5b) are based on splitting a matrix in half, in-place transposing these sub-matrices by calling `TRIP` on them, and combining the results thereafter. "Splitting a matrix in half" is formally realized using sub-matrices.

Definition 3 (Sub-matrix)**.** A ***sub-matrix*** $A(m_0 : m_1, n_0 : n_1)$ of matrix A is defined by its ranges in both dimensions, like slices in programming languages.

Let A be a matrix of dimension $m \times n$. Let $i_0, i_1, j_0, j_1 \in \mathbb{N}, 0 \le i_0 < i_1 < m$, and $0 \le j_0 < j_1 < n$. Let $B := A(i_0 : i_1, j_0 : j_1)$ be a sub-matrix of A of dimension $i_1 - i_0 \times j_1 - j_0$.

Then B is defined as follows:

$$\forall i < i_1 - i_0, j < j_1 - j_0 : b_{i,j} = a_{i_0+i, j_0+j}$$

For example,

$$A := \begin{pmatrix} a_{0,0} & a_{0,1} & \boldsymbol{a_{0,2}} & \boldsymbol{a_{0,3}} \\ a_{1,0} & a_{1,1} & \boldsymbol{a_{1,2}} & \boldsymbol{a_{1,3}} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \end{pmatrix} \quad \Rightarrow \quad A(0:2, 2:4) = \begin{pmatrix} a_{0,2} & a_{0,3} \\ a_{1,2} & a_{1,3} \end{pmatrix} \tag{3.1}$$

Inverse to the concept of the sub-matrix is the composed matrix.

Definition 4 (composed matrices)**.** Let A be a matrix of dimension $m \times n$, with $m, n \geq 1$. Let $k \in \mathbb{N}, 0 \geq k \geq n-1$.

A can be split into two sub-matrices $A_1 = A(0 : m, 0 : k)$ and $A_2 = A(0 : m, k : n)$. Then

$$B := \begin{pmatrix} A_1 & A_2 \end{pmatrix}$$

is the ***horizontally composed matrix*** of A_1 and A_2, with

$$b(i,j) := \begin{cases} a_1(i,j) & \text{if i < k} \\ a_2(i-k,j) & \text{if i >= k} \end{cases}$$

and $B = A$.

Likewise, let $l \in \mathbb{N}, 0 < l < m-1$. A can be split into two sub-matrices $A_3 = A(0 : l, 0 : n)$ and $A_4 = A(l : m, 0 : n)$. Then

$$C := \begin{pmatrix} A_3 \\ A_4 \end{pmatrix}$$

is the ***vertically composed matrix*** of A_3 and A_4, with

$$c(i,j) := \begin{cases} a_3(i,j) & \text{if j < l} \\ a_4(i,j-l) & \text{if j >= l} \end{cases}$$

and $C = A$.

Given horizontally composed matrices and vertically composed matrices, the concept of matrix composition can be extended to four or more matrices, provided their dimensions are compatible.

In order to build the bridge between computer memory, which is 1-dimensional, and matrices as defined in Definition 1, we define the array-representation of a matrix to be the mapping from a 2-dimensional matrix to a 1-dimensional array.

Definition 5 (array representation)**.** Let A be an $m \times n$ matrix.

An array $r := \overline{A}$ of length $m \times n$ is the ***array representation*** of A, if $\forall i, j : r_{n \cdot i + j} = a_{i,j}$

As an example,

$$B := \begin{pmatrix} a_0 & a_1 \\ a_2 & a_3 \end{pmatrix} \qquad \Rightarrow \qquad \overline{B} = \begin{pmatrix} a_0, a_1, a_2, a_3 \end{pmatrix}$$

Conversely, an array of length $m \cdot n$ can be interpreted as a matrix of dimensions $m \times n$ or $n \times m$.

Definition 6 (matrix interpretation)**.** Let r be an array of length $m \cdot n$.

Then $A := r^*_{m,n}$ is the matrix interpretation of r, with

$$a_{i,j} := r_{i \cdot n + j}$$

As simple corollary it follows that the $m \times n$ matrix interpretation of the array representation of an $m \times n$ matrix is the matrix itself.

Similar to sub-matrices we need the concept of sub-arrays, and their composition.

Definition 7 (sub-array)**.** Let R be an array of length n, and let $p, q \in \mathbb{N}$ with $0 \leq p < q < n$.

Then $R(p : q)$ is a sub-array of R of length $q - p$, with

$$R(p : q)_i = R_{i+p}, \qquad \forall i \in \{0, ..., q - p - 1\}$$

Conversely, the infix-concatenation of two arrays is defined as follows.

Definition 8 (infix-concatenation)**.** Let P be an array of length p and Q be an array of length q. Let $R := P \cdot Q$ be the concatenation of P and Q, an array of length $p + q$.

Then the following holds for R

$$\forall 0 \leq i < p + q : r_i = \begin{cases} p_i & \text{if } i < p \qquad (3.4a) \\ q_{i-p} & \text{if } i \geq p \qquad (3.4b) \end{cases}$$

3.2 Transpose

The transpose algorithm, `TRIP`, is the high-level recursion that is called to actually transpose a matrix. All other algorithms presented in this chapter are parts thereof.

Algorithm 3.5 defines `TRIP`. It consists of three cases. Case 3.5a if A is a tall matrix, case 3.5b if A is wide and case 3.5c, the base case, if A is square.

Definition 9 (`TRIP`)**.** Let $m, n \in \mathbb{N}$, $m, n > 0$ and A a $m \times n$ matrix (i.e. a matrix with m lines and n columns).

We define

$$\mathrm{TRIP}(A, m, n) := \begin{cases} \begin{array}{l} \text{let } B = \overline{\mathrm{TRIP}(A(0 : \lfloor \frac{m}{2} \rfloor, 0 : n), \lfloor \frac{m}{2} \rfloor, n)} \text{ and} \\ \text{let } C = \overline{\mathrm{TRIP}(A(\lfloor \frac{m}{2} \rfloor : m, 0 : n), \lceil \frac{m}{2} \rceil, n)} \text{ in} \\ \text{let } M = \mathrm{merge}\left(\overline{\begin{pmatrix} B^*_{\lfloor \frac{m}{2} \rfloor, n} \\ C^*_{\lceil \frac{m}{2} \rceil, n} \end{pmatrix}}, \lfloor \frac{m}{2} \rfloor, \lceil \frac{m}{2} \rceil, n \right) \text{ in} \\ M^*_{n,m} \end{array} & \text{if } m > n \qquad (3.5a) \\ \\ \begin{array}{l} \text{let } B = \overline{\mathrm{TRIP}(A(0 : m, 0 : \lfloor \frac{n}{2} \rfloor), m, \lfloor \frac{n}{2} \rfloor)} \text{ and} \\ \text{let } C = \overline{\mathrm{TRIP}(A(0 : m, \lfloor \frac{n}{2} \rfloor : n), m, \lceil \frac{n}{2} \rceil)} \text{ in} \\ \text{let } S = \mathrm{split}\left(\overline{\begin{pmatrix} B^*_{m, \lfloor \frac{n}{2} \rfloor} & C^*_{m, \lceil \frac{n}{2} \rceil} \end{pmatrix}}, \lfloor \frac{n}{2} \rfloor, \lceil \frac{n}{2} \rceil, m \right) \text{ in} \\ S^*_{n,m} \end{array} & \text{if } m < n \qquad (3.5b) \\ \\ \text{square_transpose}(A, n) & \text{if } m = n \qquad (3.5c) \end{cases}$$

that returns the transpose A^T of A, i.e. an $n \times m$ matrix.

Notice that the definition of *in-place*, as stated in Chapter 1, stays valid even when applied to sub-matrices. An array that represents a sub-matrix may not be contiguous in memory, but the transpose of the sub-matrix will occupy the same memory locations as the original array. Consider the example of Definition 5. In-place transposing $A(0:2,2:4)$ by applying TRIP would result in

$$A = \begin{pmatrix} a_0 & a_1 & \boldsymbol{a_2} & \boldsymbol{a_6} \\ a_4 & a_5 & \boldsymbol{a_3} & \boldsymbol{a_7} \\ a_8 & a_9 & a_{10} & a_{11} \end{pmatrix}$$

and

$$\overline{A} = (a_0, a_1, \boldsymbol{a_2}, \boldsymbol{a_6}, a_4, a_5, \boldsymbol{a_3}, \boldsymbol{a_7}, a_8, a_9, a_{10}, a_{11})$$

An efficient approach to maintaining this property in an implementation is presented in section 3.6.

The following Cilk procedure implements Algorithm 3.5 on a (sub-)matrix $A(i_0 : i_1, j_0 : j_1)$. For a matrix of dimension $m \times n$ the whole matrix is transposed when calling trip$(A, 0, m, 0, n)$.

```
cilk void trip(A, i0, i1, j0, j1) {
  m = i1 - i0;
  n = j1 - j0;
  if ((m = 1) || (n = 1)) {
    return;
  } else if (m = n) {
    spawn square_transpose(A, i0, j0, 0, m, 0, n);
    return;
  } else if (m > n) {
    im = (i1 + i0)/2;
    spawn trip(A, i0, im, j0, j1);
    spawn trip(A, im, i1, j0, j1);
    sync;
    spawn merge(A, im - i0, i1 - im, i0, i1, j0, j1);
    return;
  } else { // (m < n)
    jm = (j1 + j0)/2;
    spawn trip(A, i0, i1, j0, jm);
    spawn trip(A, i0, i1, jm, j1);
    sync;
    spawn split(A, jm - j0, j1 - jm, i0, i1, j0, j1);
    return;
  }
  return;
}
```

While it's not apparent in the definition, this algorithm is highly parallel.

In the code above the recursive calls to TRIP on the sub-matrices of A can be executed in parallel, since all operations in TRIP, merge and split act only within the bounds of their sub-matrices. The subsequent

calls to `merge` or `split` (case 3.5a or 3.5b) need to be called in sequence to the recursive calls on `TRIP`, since they combine the results of `TRIP`. Finally, in a concrete implementation of this algorithm, the matrix interpretation of the resulting array ($M^*_{n,m}$ or $S^*_{n,m}$) is done simply by means of swapping the row and column count in the meta-data of the matrix.

The following sections describe the sub-routines of `TRIP`: `merge`, `split`, and `square_transpose`.

3.3 Merge

Calling `TRIP` recursively on the two sub-matrices of a vertically partitioned, tall ($m > n$) matrix in-place transposes these sub-matrices. The subsequent `merge` call combines these two results, and transforms the transposed sub-matrices into the transpose of the full matrix.

Algorithm 3.6 defines `merge`, followed by an example.

Definition 10 (`merge`)**.** Let R be an array-representation of a (sub-)matrix of dimension $m \times n$, and $p, q \in \mathbb{N}$ with $p + q = m$.

We define

$$\text{merge}(R, p, q, n) := \begin{cases} \text{let } S = \text{rol}(R(\lfloor \frac{n}{2} \rfloor p : np + \lfloor \frac{n}{2} \rfloor q), \lceil \frac{n}{2} \rceil p) \text{ in} \\ \text{let } T_1 = \text{merge}(S(0 : \lfloor \frac{n}{2} \rfloor (p+q)), p, q, \lfloor \frac{n}{2} \rfloor) \text{ and} \\ \text{let } T_2 = \text{merge}(S(\lfloor \frac{n}{2} \rfloor (p+q) : n(p+q)), p, q, \lceil \frac{n}{2} \rceil) \text{ in} & \text{if } n > 1 \quad (3.6a) \\ T_1 \cdot T_2 \\ \\ R & \text{if } n = 1 \quad (3.6b) \end{cases}$$

that combines the array-representation of two transposed sub-matrices (R) to the array-representation of the transpose of the vertically composed matrix of these two sub-matrices.

`merge` consists of two cases, depending on the recursion parameter n. Case 3.6a first rotates the inner part of array $\overline{A}$ left, after which `merge` recurses in parallel on the left and right sub-arrays. Case 3.6b is the base case of the recursion.

Example `TRIP` is called on a tall matrix A. `merge` will be called after the recursive transposition of the upper and lower half sub-matrices (c.f. Algorithm 3.5). The `merge` recursion is executed in this example, showing data movements and the call graph.

$$A := \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \\ 13 & 14 & 15 \end{pmatrix}$$

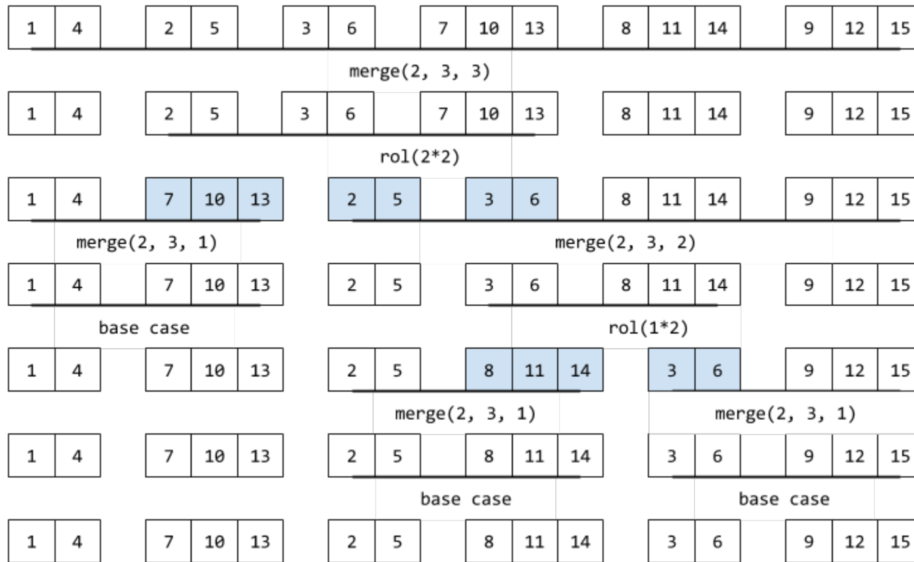
`TRIP` in-place transposes the sub-matrices $A(0 : 2, 0 : 3)$ and $A(2 : 5, 0 : 3)$ in-place, resulting in the following setup for the subsequent call to `merge`

n = 3

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

m = 5, p = 2, q = 3

The merge recursion first rotates the center part of the array, then recursively calls itself. This merges the two parts of the array, the in-place transpose of the upper sub-matrix (left, blocks of two), and the in-place transpose of the lower sub-matrix (right, blocks of three). The block-length is set to p and q to visualize the original position of the elements, and to motivate the name of the procedure. Blocks of length p and q are "merged" by `merge`. Highlighted cells indicate moved elements.



The resulting array can be interpreted as A^T:

$$\begin{pmatrix} 1 & 4 & 7 & 10 & 13 \\ 2 & 5 & 8 & 11 & 14 \\ 3 & 6 & 9 & 12 & 15 \end{pmatrix} = A^T$$

The following Cilk procedure implements Algorithm 3.6 on a (sub-)matrix $A(i_0 : i_1, j_0 : j_1)$, that has been divided into two sub-matrices with p and q rows each. s_0, s_1 and n are recursion parameters. Initially `merge` is called with $s_0 = 0$ and $s_1 = (i_1 - i_0) \cdot (j_1 - j_0)$, i.e. the array boundaries s_0 and s_1 cover the whole array-representation of the sub-matrix.

```
cilk void merge(A, p, q, i0, i1, j0, j1, s0, s1, n) {
  if (n = 1)
    return;
```

```
    r_0 = s_0 + (n/2) · p;
    r_1 = s_0 + n · p + (n/2) · q;
    spawn rol(A(r_0 : r_1), (n/2) · p, i_0, i_1, j_0, j_1);
    sync;

    s_m = s_0 + (n/2) · (p + q);
    spawn merge(A, p, q, i_0, i_1, j_0, j_1, s_0, s_m, (n/2));
    spawn merge(A, p, q, i_0, i_1, j_0, j_1, s_m, s_1, n − (n/2));
    return;
}
```

While it's not apparent in the definition, this algorithm is highly parallel. In the code above the recursive calls to `merge` on the sub-arrays of S can be executed in parallel, since all operations in `merge` act only within the bounds of their sub-arrays. The preceding call to `rol` needs to be done in sequence to the recursive calls on `merge`, since `rol` modifies parts of both sub-arrays that are then passed to `merge` calls.

Summarizing, the procedure `merge` shuffles groups of elements of two previously transposed, vertically stacked sub-matrices to transform them from being vertically stacked, to be horizontally aligned. Blocks of the top sub-matrix are moved to the left of the new transposed matrix, blocks of the bottom sub-matrix are moved to the right of the transposed matrix. In the array representation of the matrix that corresponds to a perfect shuffle its array blocks.

3.4 Split

Calling `TRIP` recursively on the two sub-matrices of a horizontally partitioned, wide ($m < n$) matrix in-place transposes these sub-matrices. The subsequent `split` call combines these two results, and transforms the transposed sub-matrices to the transpose of the full matrix.

Algorithm 3.7 defines `split`. An example illustrates the data-flow after the definition.

Definition 11 (`split`)**.** Let R be an array-representation of a (sub-)matrix of dimension $m \times n$, and $p, q \in \mathbb{N}$ with $p + q = n$.

We define

$$\text{split}(R, p, q, m) := \begin{cases} \text{let } S_1 = \text{split}(R(0 : \lfloor \frac{m}{2} \rfloor (p+q)), p, q, \lfloor \frac{m}{2} \rfloor) \text{ and} \\ \text{let } S_2 = \text{split}(R(\lfloor \frac{m}{2} \rfloor (p+q) : m(p+q)), p, q, \lceil \frac{m}{2} \rceil) \text{ in} \\ \text{let } T = S_1 \cdot S_2 \text{ in} \\ \text{rol}(T(\lfloor \frac{m}{2} \rfloor p : mp + \lfloor \frac{m}{2} \rfloor q), \lfloor \frac{m}{2} \rfloor q) & \text{if } m > 1 \quad (3.7a) \\ \\ R & \text{if } m = 1 \quad (3.7b) \end{cases}$$

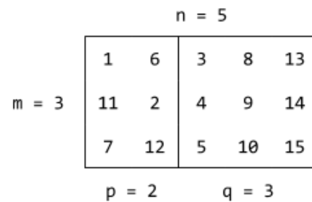
that combines the array-representation of two transposed sub-matrices (R) to the array-representation of the transpose of the horizontally composed matrix of these two sub-matrices.

`split` contains a recursive case and a base case. Unlike `merge` the left rotation follows after the recursive calls of `split`. The symmetry of `merge` and `split` is the result of the two functions being inverse.

Example For this example `TRIP` is called on a wide matrix A. `split` will be called after the recursive transposition of the left and right sub-matrices (c.f. Algorithm 3.5). The `split` recursion is executed here, showing data movements and call graph. Highlighted cells indicate moved elements.

$$A := \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \end{pmatrix}$$

`TRIP` in-place transposes the sub-matrices $A(0:3, 0:2)$ and $A(0:3, 2:5)$, resulting in the following setup for the subsequent call to `split`:



The length of the blocks highlights the original sub-matrix of the elements. The split recursion, when collapsing the `split`-tree, rotates the blocks of length q to the right, effectively "splitting" p blocks and q blocks.

The resulting array can be interpreted as the transpose of the initial matrix:

$$\begin{pmatrix} 1 & 6 & 11 \\ 2 & 7 & 12 \\ 3 & 8 & 13 \\ 4 & 9 & 14 \\ 5 & 10 & 15 \end{pmatrix} = A^T$$

While `merge` alternates blocks of the sub-matrices to horizontally align them, `split` separates them, to transform the horizontally aligned sub-matrices of a wide matrix to vertically stacked sub-matrices of a tall matrix-transpose.

The following Cilk procedure implements Algorithm 3.7 on a (sub-)matrix $A(i_0 : i_1, j_0 : j_1)$, that has been divided into two sub-matrices with p and q columns each.

```
cilk void split(A, p, q, i_0, i_1, j_0, j_1, s_0, s_1, m) {
  if (m = 1)
    return;

  s_m = s_0 + (m/2) · (p + q);
  spawn split(A, p, q, i_0, i_1, j_0, j_1, s_0, s_m, (m/2));
  spawn split(A, p, q, i_0, i_1, j_0, j_1, s_m, s_1, m − (m/2));
  symc;

  r_0 = s_0 + (m/2) · p;
  r_1 = s_0 + m · p + (m/2) · q;
  spawn rol(A(r_0 : r_1), (m/2) · q, i_0, i_1, j_0, j_1);
  return;
}
```

As with `merge`, this algorithm is highly parallel.
In the code above the recursive calls to `split` on the sub-arrays of R can be executed in parallel, since all operations in `split` act only within the bounds of their sub-arrays. The subsequent call to `rol` needs to be done in sequence to the recursive calls on `split`, since `rol` modifies parts of both sub-arrays.

The next section describes the base case of `TRIP`, a transposition algorithm for square matrices.

3.5 Square Transpose

This novel, highly parallel algorithm is the base case of the `TRIP` recursion. Like `TRIP` it is recursive in nature, and in-place transposes square matrices.

Definition 12 (swap)**.** Let A be a square matrix of dimension $n \times n$.

We define

$$\mathrm{swap}(A, i_0, i_1, j_0, j_1) := B \text{ where } b(j - j_0, i - i_0) = a(i, j), \forall j \in \{j_0, ...j_1\}, i \in \{i_0, ..., i_1\}$$

that returns a matrix of dimension $j_1 - j_0 \times i_1 - i_0$ that is the sub-matrix $A(i_0 : i_1, j_0 : j_1)$, but mirrored across the diagonal.

Using this, we define algorithm `square_transpose`.

Definition 13 (`square_transpose`)**.** Let A be a square matrix of dimension $n \times n$.

We define

$$\text{sq}(A, i_0, i_1, j_0, j_1) := \begin{cases} \begin{array}{l} \text{let } i_m = \lfloor \frac{i_0+i_1}{2} \rfloor \text{ and} \\ \text{let } j_m = \lfloor \frac{j_0+j_1}{2} \rfloor \text{ in} \\ \text{let } B = \text{sq}(A, i_0, i_m, j_0, j_m) \text{ and} \\ \text{let } C = \text{sq}(A, i_0, i_m, j_m, j_1) \text{ and} \\ \text{let } D = \text{sq}(A, i_m, i_1, j_0, j_m) \text{ and} \\ \text{let } E = \text{sq}(A, i_m, i_1, j_m, j_1) \text{ in} \\ \begin{pmatrix} B & C \\ D & E \end{pmatrix} \end{array} & \text{if } i_1 - i_0 > 1 \land i_1 < j_0 \quad (3.8a) \\ \begin{array}{l} \text{let } i_m = \lfloor \frac{i_0+i_1}{2} \rfloor \text{ and} \\ \text{let } j_m = \lfloor \frac{j_0+j_1}{2} \rfloor \text{ in} \\ \text{let } B = \text{sq}(A, i_0, i_m, j_0, j_m) \text{ and} \\ \text{let } C = \text{sq}(A, i_0, i_m, j_m, j_1) \text{ and} \\ \text{let } D = \text{swap}(A, j_0, j_m, i_m, i_1) \text{ and} \\ \text{let } E = \text{sq}(A, i_m, i_1, j_m, j_1) \text{ in} \\ \begin{pmatrix} B & C \\ D & E \end{pmatrix} \end{array} & \text{if } i_1 - i_0 > 1 \land i_1 \geq j_0 \quad (3.8b) \\ \text{swap}(A, j_0, j_m, i_m, i_1) & \text{else} \quad (3.8c) \end{cases}$$

that returns the transposed matrix of square-matrix A.

This recurses to single rows and a maximum number of two columns swaps those entries with the corresponding entries below the diagonal, and returns.

The algorithm consists of two cases and a base case.
Case 3.8a is met for sub-matrices that are fully above the diagonal of A. In this case the algorithm recurses all the way to the base case 3.8c, and swaps the elements with those below the diagonal. Case 3.8b is met for sub-matrices that are on or below the diagonal, including the root call of `square_transpose`, where the sub-matrix is A itself. Here, the upper-left, upper-right and lower-right sub-matrices are treated as in Case 3.8a, i.e. the algorithm recurses all the way to the base case, where the elements are swapped. The lower-left sub-matrix however is swapped directly.

When implemented in an imperative language, this distinction between Cases 3.8a and Case 3.8b disappears, since a single "swap" can modify both entries: the one above the diagonal, and the one below the diagonal.

The following Cilk procedure implements Algorithm 3.8 on a (sub-)matrix A of dimension $i_1 - i_0 \times j_1 - j_0$.

```
cilk square_transpose(A, i0, i1, j0, j1) {
  if (i1 − i0 > 1) {
    im = (i0 + i1)/2;
    jm = (j0 + j1)/2;
    spawn square_transpose(A, i0, im, j0, jm);
    spawn square_transpose(A, i0, im, jm, j1);
    spawn square_transpose(A, im, i1, jm, j1);
    if (i1 ≤ j0)
      spawn square_transpose(A, im, i1, j0, jm);
  } else {
    for (j = j0; j < j1; j++) {
      swap(A[j,i0], A[i0,j]);
    }
  }
}
```

The square matrix A is recursively split into sub-matrices, until, in the base case, the sub-matrix contains only one or two elements. Entries of sub-matrices above the diagonal are swapped with entries below the diagonal in the base case. Since the base case is only called for entries above the diagonal, all recursive calls are independent of each other. This means for each recursion, the parallelism increases by a factor of three to four.

Any efficient, parallel, in-place square transpose algorithm can be used at this place, and TRIP will work.

All presented algorithms have to accept sub-matrices as input. The relative inefficiency of modulo and divide operations in comparison to addition or multiplication makes index calculations relevant in this context. The next section will describe iterators that are key to an efficient implementation of TRIP.

3.6 Iterators

Iterators efficiently calculate the index of consecutive entries of a sub-matrix, in the index-system of the original matrix, as opposed to a simple mapping.

For a matrix $A_{M,N}$ and a sub-matrix $A(i_0 : i_1, j_0 : j_1)$ this mapping from sub-matrix array indices to matrix array indices could be achieved by the mapping

$$i \quad \rightarrow \quad \left(i_0 + \lfloor i/N_s \rfloor\right) \cdot N + j_0 + (i \mod N_s)$$

where $N_s = j_1 - j_0$ is the second dimension of the sub-matrix. While simple, this mapping involves the operations division and modulo. Since implementations of the discussed algorithms make heavy use

of sub-matrices, iterators, as an efficient method to transform the index systems of sum-matrices to the index systems of their original matrices, are necessary:

Iterators depend on an internal state, in this case the variable `count`, that keeps track of the position inside the sub-matrix, and triggers an index-jump into the next row of the original matrix, if necessary. Using iterators instead of a naive mapping decreases the computational time needed for index calculations dramatically.

The forward iterator for sub-matrix $A(i_0 : i_1, j_0 : j_1)$ updates i, an index of A, to point to the next element in the sub-matrix.

```
next(*i, *count, j0, j1, N, Ns) {
  if (*count == Ns - 1) {
    *count = 0;
    *i += (N - j1) + j0 + 1;
  } else {
    *count += 1;
    *i += 1;
  }
}
```

The corresponding reverse iterator, which updates i to the index of the previous element of $A(i_0 : i_1, j_0 : j_1)$ within A is based on the same principle:

```
prev(*i, *count, j0, j1, N, Ns) {
  if (*count == 0) {
    *count = Ns - 1;
    *i -= j0 + (N - j1) + 1;
  } else {
    *count -= 1;
    *i -= 1;
  }
}
```

The next chapter provides proofs of correctness for algorithms: `TRIP`, `merge`, and `split`.

The full implementation of TRIP can be found in Appendices 1 and 2 in the form of Cilk code that can be complied with e.g. gcc-7.3.

4 Proof of Correctness

A proof of the correctness of algorithm TRIP is presented in this chapter. It consists of three parts. The first part (Section 4.1) proves the correctness of the `merge` algorithm (see 3.3). The second part (Section 4.2) proves the correctness of the `split` algorithm (see 3.4). The third part (Section 4.3) integrates the first two parts and proves correctness of the `TRIP` recurrence as a corollary.

4.1 Correctness Proof of Combine Method `merge`

In this section we prove correctness of the `merge` algorithm for *tall* matrices of dimension $M \times N$ with $M > N$. We prove that given two vertically aligned sub-matrices T_p and T_q that are in-place transposes of sub-matrices that partition A, `merge` will combine these two sub-matrices to form $A^\top$.

The first part of this section introduces the structure of the input of `merge`. The second part consists of theorem and proof.

In `TRIP` two sub-matrices of a matrix A always partition that matrix. Specifically, for all $p, q > 0$ with $p + q = M$, A can be partitioned like this

$$A = \underbrace{\left(\begin{array}{c} A_p \\ \\ A_q \end{array} \right)}_{N} \begin{array}{l} \}\, p \\ \\ \}\, q \end{array}$$

The transpose $A^\top$ of A is made up of the transposes of these sub-matrices as well and has the shape

$$A^\top = \left(\underbrace{A_p^\top}_{p} \quad \underbrace{A_q^\top}_{q} \right) \}\, N$$

i.e.

$$\overline{A^\top} = \prod_{0 \le i < N} \left(\left(\overline{A_p^\top} \right)_i \cdot \left(\overline{A_q^\top} \right)_i \right) \tag{4.1}$$

where

$$\prod_{0 \le i < k} \overline{A_i} = \overline{A_0} \cdot \overline{A_1} \cdot \ldots \cdot \overline{A_{k-1}}$$

is the ***prefix concatenation*** operator, and $\cdot$ is the ***infix concatenation*** operator:

$$\left(a_0, \ldots, a_m\right) \cdot \left(b_0, \ldots, b_n\right) = \left(a_0, \ldots, a_m, b_0, \ldots, b_n\right)$$

$\left(\overline{A_p}\right)_i$ is the i-th ***subarray*** of array-size p of the array representing matrix A_p. For example if

$$A_2 = \begin{pmatrix} 4 & 5 & 9 \\ 10 & 14 & 15 \end{pmatrix}$$

then $\left(\overline{A_2}\right)_0 = (4, 5)$, $\left(\overline{A_2}\right)_1 = (9, 10)$ and $\left(\overline{A_2}\right)_2 = (14, 15)$.

`merge` is *not* applied directly to A (i.e. to A_p and A_q) though. Since A is a tall matrix, `TRIP` (Algorithm 3.5) recursively applies itself to $A_p = A(0 : \lfloor\frac{M}{2}\rfloor, 0 : N)$ and $A_q = A(\lceil\frac{M}{2}\rceil : M, 0 : N)$, returning a matrix that is composed of two in-place transposed sub-matrices. After the in-place transposition of A_p and A_q `merge` combines the result. In-place transposed sub-matrices can be written as ***reshaped transposes*** T_p and T_q where T_p is the $p \times N$ matrix that fulfills

$$\overline{T_p} = \overline{A_p^\top}$$

and T_q is the $q \times N$ matrix that fulfills

$$\overline{T_q} = \overline{A_q^\top}$$

Applying `TRIP` to A_p and A_q results in a matrix T of dimension $(p + q) \times N$ with

$$T = \underbrace{\begin{pmatrix} T_p \\ \\ T_q \end{pmatrix}}_{N} \begin{matrix} \}\, p \\ \\ \}\, q \end{matrix}$$

i.e., T is composed as follows:

$$\overline{T} = \overline{T_p} \cdot \overline{T_q} = \overline{A_p^\top} \cdot \overline{A_q^\top} = \prod_{0 \le i < N} \left(\overline{A_p^\top}\right)_i \cdot \prod_{0 \le i < N} \left(\overline{A_q^\top}\right)_i \tag{4.2}$$

`merge` transforms T into $A^\top$: In order to derive $\overline{A^\top}$ (Equation 4.1) from $\overline{T}$, merge "merges" $\overline{T_p}$ and $\overline{T_q}$, i.e. it intertwines consecutive sub-arrays of $\overline{A_p^\top}$ and $\overline{A_q^\top}$ in $\overline{T}$.

Lemma 1. $\forall M, N \in \mathbb{N}, \forall p, q > 0$ *with* $p + q = M$
Let A be a matrix of dimension $M \times N$, and $\overline{T_p} = \overline{A_p^\top}, \overline{T_q} = \overline{A_q^\top}, \overline{T} = \overline{T_p} \cdot \overline{T_q}$.
Then

$$merge(\overline{T}, p, q, N) = \overline{A^\top}$$

Proof. We prove this for all N by assuming arbitrary but fixed $M, p, q \in \mathbb{N}$ with $p + q = M$ and A a matrix with dimensions $M \times N$, and performing induction over N.

Base Case $N = 1$. N is the number of columns in A, and due to Equation 4.2 also the number of sub-array-pairs in $\overline{T}$, i.e. pairs $\left(\overline{A_p^\top}\right)_i, \left(\overline{A_q^\top}\right)_i, \forall i \in 0, \dots, N-1$.

Due to equations 4.2 and 4.1, and `merge` matching the base case, we have

$$\overline{T} = \left(\overline{A_p^\top}\right)_0 \cdot \left(\overline{A_q^\top}\right)_0 = \overline{A_p^\top} \cdot \overline{A_q^\top} = \prod_{0 \le i < N} \left(\left(\overline{A_p^\top}\right)_i \cdot \left(\overline{A_q^\top}\right)_i\right) = \overline{A^\top}$$

which proves the base case.

Induction Hypothesis Let $N_0 \ge 1$ be arbitrary but fixed. We assume for all $k \le N_0$

$$\texttt{merge}(\overline{T}, p, q, k) = \overline{A^\top}$$

Induction Step When calling merge$(\overline{T}, p, q, N_0 + 1)$, we have

$$\overline{T} = \prod_{0 \le i < N_0+1} \left(\overline{A_p^\top}\right)_i \cdot \prod_{0 \le i < N_0+1} \left(\overline{A_q^\top}\right)_i$$

Since $N_0 + 1 > 1$ this matches the second case of the recursion.

The first operation in this case is rol$(\overline{T}(\lfloor\frac{N_0+1}{2}\rfloor p : (N_0+1)p + \lfloor\frac{N_0+1}{2}\rfloor q), \lceil\frac{N_0+1}{2}\rceil p)$ which results in

$$\overline{T} = \prod_{0 \le i < \lfloor\frac{N_0+1}{2}\rfloor} \left(\overline{A_p^\top}\right)_i \cdot \prod_{0 \le i < \lfloor\frac{N_0+1}{2}\rfloor} \left(\overline{A_q^\top}\right)_i \cdot \prod_{\lfloor\frac{N_0+1}{2}\rfloor \le i < N_0+1} \left(\overline{A_p^\top}\right)_i \cdot \prod_{\lfloor\frac{N_0+1}{2}\rfloor \le i < N_0+1} \left(\overline{A_q^\top}\right)_i$$

Second, we apply merge$(\overline{T}(0 : \lfloor\frac{N_0+1}{2}\rfloor(p+q)), p, q, \lfloor\frac{N_0+1}{2}\rfloor)$. This allows to use the induction hypothesis, since the subarray $\overline{T}(0 : \lfloor\frac{N_0+1}{2}\rfloor(p+q))$ has the correct shape and the number of subarray-pairs is $\lfloor\frac{N_0+1}{2}\rfloor \le N_0$. Thus

$$\overline{T} = \prod_{0 \le i < \lfloor\frac{N_0+1}{2}\rfloor} \left(\left(\overline{A_p^\top}\right)_i \cdot \left(\overline{A_q^\top}\right)_i\right) \cdot \prod_{\lfloor\frac{N_0+1}{2}\rfloor \le i < N_0+1} \left(\overline{A_p^\top}\right)_i \cdot \prod_{\lfloor\frac{N_0+1}{2}\rfloor \le i < N_0+1} \left(\overline{A_q^\top}\right)_i$$

When (in parallel) applying merge$(\overline{T}(\lfloor\frac{N_0+1}{2}\rfloor(p+q) : (N_0+1)(p+q)), p, q, \lceil\frac{N_0+1}{2}\rceil)$, the induction hypothesis can be used again, since the subarray $\overline{T}(\lfloor\frac{N_0+1}{2}\rfloor(p+q) : (N_0+1)(p+q))$ too has the correct shape and $\lceil\frac{N_0+1}{2}\rceil \le N_0$. Now

$$\begin{aligned}\overline{T} &= \prod_{0 \le i < \lfloor\frac{N_0+1}{2}\rfloor} \left(\left(\overline{A_p^\top}\right)_i \cdot \left(\overline{A_q^\top}\right)_i\right) \cdot \prod_{\lfloor\frac{N_0+1}{2}\rfloor \le i < N_0+1} \left(\left(\overline{A_p^\top}\right)_i \cdot \left(\overline{A_q^\top}\right)_i\right) \\ &= \prod_{0 \le i < N_0+1} \left(\left(\overline{A_p^\top}\right)_i \cdot \left(\overline{A_q^\top}\right)_i\right) = \overline{A^\top}\end{aligned}$$

as can be seen by comparison with equation 4.1.

□

The next section contains the correctness proof of `split`. Since `merge` and `split` are inverse to each other, the proofs are similar.

4.2 Correctness Proof of Combine Method `split`

In this section we prove correctness of the `split` algorithm for *wide* matrices of dimension $M \times N$ with $M < N$. We prove that given two horizontally aligned sub-matrices T_p and T_q that are in-place transposes of sub-matrices that partition A, `split` will combine these two sub-matrices to form $A^\top$.

The first part of this section introduces the structure of the input of `split`. The second part consists of theorem and proof.

Again, `TRIP` calls `split` on sub-matrices of A that partition A. Specifically, with $p, q > 0$ and $p + q = N$ it can be partitioned like this

$$A = \left.\left(\underbrace{\quad A_p \quad}_{p} \quad \underbrace{\quad A_q \quad}_{q} \right)\right\} M$$

The transpose $A^\top$ of A is made up of the transposes of these sub-matrices as well, and has the shape

$$A^\top = \underbrace{\begin{pmatrix} A_p^\top \\ \\ A_q^\top \end{pmatrix}}_{M} \begin{matrix} \}\, p \\ \\ \}\, q \end{matrix}$$

i.e.

$$\overline{A^\top} = \overline{A_p^\top} \cdot \overline{A_q^\top} = \left(\prod_{0 \le i < M} \left(\overline{A_p^\top} \right)_i \right) \cdot \left(\prod_{0 \le i < M} \left(\overline{A_q^\top} \right)_i \right) \tag{4.3}$$

`split` is *not* applied directly to A (i.e. to A_p and A_q). Since A is a wide matrix, `TRIP` (Algorithm 3.5) recursively applies itself to $A_p = A(0 : M, 0 : \lfloor \frac{N}{2} \rfloor)$ and $A_q = A(0 : M, \lceil \frac{N}{2} \rceil : N)$, returning a matrix that is composed of two in-place transposed sub-matrices. After the in-place transposition of A_p and A_q, `split` combines the result. In-place transposed sub-matrices can be written as ***reshaped transposes*** T_p and T_q where T_p is the $M \times p$ matrix that fulfills

$$\overline{T_p} = \overline{A_p^\top}$$

and T_q is the $M \times q$ matrix that fulfills

$$\overline{T_q} = \overline{A_q^\top}$$

Applying `TRIP` to A_p and A_q results in a matrix T of dimension $m \times (p + q)$ with

$$T = \left.\left(\underbrace{\quad T_p \quad}_{p} \quad \underbrace{\quad T_q \quad}_{q} \right)\right\} M$$

or in array representation

$$\overline{T} = \prod_{0\le i<M} \left(\left(\overline{T_p}\right)_i \cdot \left(\overline{T_q}\right)_i\right) = \prod_{0\le i<M} \left(\left(\overline{A_p^\top}\right)_i \cdot \left(\overline{A_q^\top}\right)_i\right) \tag{4.4}$$

or expanded:

$$\overline{T} = \left(\overline{A_p^\top}\right)_0 \cdot \left(\overline{A_q^\top}\right)_0 \cdot \left(\overline{A_p^\top}\right)_1 \cdot \left(\overline{A_q^\top}\right)_1 \cdot \ldots \cdot \left(\overline{A_p^\top}\right)_{M-1} \cdot \left(\overline{A_q^\top}\right)_{M-1}$$

`split` transforms $\overline{T}$ into $\overline{A^\top}$: In order to derive $\overline{A^\top}$ (equation 4.3) from $\overline{T}$, split "splits" (i.e. separates) sub-arrays of $\overline{A_p^\top}$ and $\overline{A_q^\top}$ in $\overline{T}$.

Lemma 2. $\forall M, N \in \mathbb{N}, \forall p, q > 0$ *with* $p + q = N$

Let A *be a matrix of dimension* $M \times N$*, and* $\overline{T_p} = \overline{A_p^\top}, \overline{T_q} = \overline{A_q^\top}$*, and*

$$\overline{T} = \prod_{0\le i<M} \left(\left(\overline{T_p}\right)_i \cdot \left(\overline{T_q}\right)_i\right)$$

as described in Equation 4.4. Then

$$split(\overline{T}, p, q, M) = \overline{A^\top}$$

Proof. We prove this for all M by assuming arbitrary but fixed $N, p, q \in \mathbb{N}$ with $p + q = N$ and A a matrix with dimensions $M \times N$, and performing induction over M.

Base Case $M = 1$, where M is the number of rows in A, and due to Equation 4.4 also the number of sub-array-pairs in $\overline{T}$, i.e. pairs $\left(\overline{A_p^\top}\right)_i, \left(\overline{A_q^\top}\right)_i, \forall i \in 0, \ldots, M-1$.

Due to equation 4.4 and `split` matching the base case, we have

$$\overline{T} = \left(\overline{A_p^\top}\right)_0 \cdot \left(\overline{A_q^\top}\right)_0 = \overline{A_p^\top} \cdot \overline{A_q^\top} = \prod_{0\le i<k} \left(\overline{A_p^\top}\right)_i \cdot \prod_{0\le i<k} \left(\overline{A_q^\top}\right)_i = \overline{A^\top}$$

which proves the base case.

Induction Hypothesis Let $M_0 > 1$ be arbitrary but fixed. We assume for all $k \le M_0$

$$\texttt{split}(\overline{T}, p, q, k) = \overline{A^\top}$$

Induction Step When calling split$(\overline{T}, p, q, M_0 + 1)$, we have

$$\overline{T} = \prod_{0\le i<M_0+1} \left(\left(\overline{A_p^\top}\right)_i \cdot \left(\overline{A_q^\top}\right)_i\right)$$

Since $M_0 + 1 > 1$ this matches the second case of the recursion.

The first operation in this case is split$(\overline{T}(0 : \lfloor\frac{M_0+1}{2}\rfloor(p+q)), p, q, \lfloor\frac{M_0+1}{2}\rfloor)$. The induction hypothesis holds, since the subarray $\overline{T}(0 : \lfloor\frac{M_0+1}{2}\rfloor(p+q))$ has the correct shape and the number of subarray-pairs is $\lfloor\frac{M_0+1}{2}\rfloor \le M_0$. Thus

$$\overline{T} = \prod_{0\le i<\lfloor\frac{M_0+1}{2}\rfloor} \left(\overline{A_p^\top}\right)_i \cdot \prod_{0\le i<\lfloor\frac{M_0+1}{2}\rfloor} \left(\overline{A_q^\top}\right)_i \cdot \left(\prod_{\lfloor\frac{M_0+1}{2}\rfloor\le i<M_0+1} \left(\left(\overline{A_p^\top}\right)_i \cdot \left(\overline{A_q^\top}\right)_i\right)\right)$$

When in parallel applying split($\overline{T}\lfloor\frac{M_0+1}{2}\rfloor(p+q) : (M_0+1)(p+q), p, q, \lceil\frac{M_0+1}{2}\rceil$) the induction hypothesis can be used again, since the subarray $\overline{T}(\lfloor\frac{M_0+1}{2}\rfloor(p+q) : (M_0+1)(p+q))$ has the correct shape and the number of subarray-pairs is $\lceil\frac{M_0+1}{2}\rceil \leq M_0$. After this step

$$\overline{T} = \prod_{0\leq i<\lfloor\frac{M_0+1}{2}\rfloor} \left(\overline{A_p^\top}\right)_i \cdot \prod_{0\leq i<\lfloor\frac{M_0+1}{2}\rfloor} \left(\overline{A_q^\top}\right)_i \cdot \prod_{\lfloor\frac{M_0+1}{2}\rfloor\leq i<M_0+1} \left(\overline{A_p^\top}\right)_i \cdot \prod_{\lfloor\frac{M_0+1}{2}\rfloor\leq i<M_0+1} \left(\overline{A_q^\top}\right)_i \tag{4.5}$$

Second, we apply rol($\overline{T}(\lfloor\frac{M_0+1}{2}\rfloor p : (M_0+1)p + \lfloor\frac{M_0+1}{2}\rfloor q), \lfloor\frac{M_0+1}{2}\rfloor q$). This swaps the two middle parts of $\overline{T}$ in equation 4.5.

Thus finally we have

$$\begin{aligned}\overline{T} &= \prod_{0\leq i<\lfloor\frac{M_0+1}{2}\rfloor} \left(\overline{A_p^\top}\right)_i \cdot \prod_{\lfloor\frac{M_0+1}{2}\rfloor\leq i<M_0+1} \left(\overline{A_p^\top}\right)_i \cdot \prod_{0\leq i<\lfloor\frac{M_0+1}{2}\rfloor} \left(\overline{A_q^\top}\right)_i \cdot \prod_{\lfloor\frac{M_0+1}{2}\rfloor\leq i<M_0+1} \left(\overline{A_q^\top}\right)_i \\ &= \left(\prod_{0\leq i<M_0+1} \left(\overline{A_p^\top}\right)_i\right) \cdot \left(\prod_{0\leq i<M_0+1} \left(\overline{A_q^\top}\right)_i\right) = \overline{A^\top}\end{aligned}$$

as can be seen by comparison with equation 4.3.

□

The next section contains the correctness proof of `TRIP`, which follows from the correctness of `merge` and `split`.

4.3 Correctness Proof of `TRIP`

The last two sections proved the correctness of the combine methods of TRIP, depending on whether the matrix that is transposed is wide or tall. This section proves the correctness of `TRIP` itself, using the previous results.

Theorem 1. *For all matrices A with dimension $M \times N$:*

$$\mathit{TRIP}(A, M, N) = A^\top$$

Proof. Proceed by induction on the number of elements $E := M \cdot N$ of matrix A.

Base Case $E = 1$. The matrix contains one element and is of dimension 1×1. This means it is square and

$$\text{TRIP}(A, 1, 1) = \text{sq}(A, 0, 1, 0, 1) = A = A^\top$$

Induction Hypothesis Take E_0 arbitrary but fixed, for all matrices with dimension $M \times N$ such that $M \cdot N \leq E_0$ and assume

$$\text{TRIP}(A, M, N) = A^\top$$

Induction Step Matrix A is of dimension $M \times N$ and has $E_0 + 1$ elements, i.e., $M \cdot N = E_0 + 1$. We proceed by case distinction on whether the A is square, wide or tall.

Case $M = N$ The matrix is square, and we have

$$\text{TRIP}(A, M, N) = \text{sq}(A, 0, M, 0, N) = A^\top$$

Case $M > N$ The matrix is tall and `TRIP` matches Case 3.5a, i.e., A is divided in two sub-matrices with dimensions $p \times N$ and $q \times N$. Since $p \cdot N \leq E_0$ and $q \cdot N \leq E_0$, the induction hypothesis can be applied to both cases. Due to Lemma 1, `merge` subsequently combines the results to $A^\top$.

Case $M < N$ The matrix is wide and `TRIP` matches Case 3.5b, i.e. A is divided in two sub-matrices with dimensions $M \times p$ and $M \times q$ with $p = \lfloor \frac{N}{2} \rfloor$ and $q = \lceil \frac{N}{2} \rceil$. `TRIP` is applied to both sub-matrices. Since $M \cdot p \leq E_0$ and $M \cdot q \leq E_0$ the induction hypothesis can be applied to both cases. Due to Lemma 2, `split` subsequently combines the results to $A^\top$.

□

This concludes the correctness proof of `TRIP`. The next chapter analyzes the computational complexity of `TRIP` by counting work, span and parallelism in the special case of matrices, whose dimensions are powers of two.

5 Complexity Analysis

The complexity analysis of TRIP consists of four parts: First the analysis of work W_0, second the analysis of span W_∞; the third part is an analysis of the parallelism of the algorithm, and finally, the chapter is concluded by visualizing work and span as a function of the shape of the matrix and interpreting the resulting curves.

The complexity analysis of `TRIP` is deliberately restricted to matrices $A_{M,N}$ whose dimensions M and N are for simplicity powers of two. Tall matrices $A_{M,N}$ as well as wide matrices $A_{N,M}$ thus fulfill the following condition:

$$\left(\exists k \in \ N : N = 2^k\right) \wedge \left(\exists l \in \ N : M = 2^l N\right) \tag{5.1}$$

`TRIP` behaves specially when called on matrices with an aspect ratio that is a power of two: If the matrix in question is tall, the recursion keeps entering `merge` (Case 3.5a), until the sub-matrices become square (Case 3.5c). If the matrix is wide, the recursion keeps entering `split` (Case 3.5b) until the sub-matrices become square. This allows to calculate closed form solutions of work and span, since it leads to a high degree of symmetry in the recursive calls.

The following section introduces work W_1 and analyzes the work of `TRIP` and its underlying algorithms.

5.1 Work

In this section, after a brief definition of the work of a computation, Lemmas 3 and 4 cover the work of algorithms `reverse` and `rol`, which are the basis of all following calculations. After that, Theorem 2 shows that the transpose of a tall matrix A with dimensions $M \times N$ has work $\Theta\left(MN\left(1 + \log\frac{M}{N}\log N\right)\right)$, provided M and N fulfill Condition 5.1. Corollary 1 generalizes this result to wide and square matrices.

In the context of the dag model described in [5, p.777ff], the ***work*** W_1 is defined as the number of vertices in a computation dag. Our analysis considers as vertices instruction-groups, i.e.

- inner nodes in a spawned tree structure, and
- swaps.

Each inner node in a recursive call tree, which most of the time amounts to about three actual function calls, is considered one unit of work, as is a swap of two array entries.

The work analysis of two algorithms that form the base of `TRIP` is a good introduction to the general principles applied throughout this section.

Work of Base Algorithms

The work complexity of `reverse` and `rol` depends on the applied algorithms. The following Cilk algorithms are used in this thesis.

In order to achieve parallelism, the array reversal algorithm is recursive in nature. Slice $A(m_0 : m_1)$ is reversed, l is initialized with $(m_1 - m_0)/2$ and is the recursion parameter.

```
cilk reverse(A, m0, m1, l) {
  if (l > 1) {
    lm = l/2;
    spawn reverse(A, m0, m1, lm);
    spawn reverse(A, m0 + lm, m1 − lm, l − lm);
  } else {
    swap(A, m0, m1 − 1);
  }
}
```

To rotate array A we apply Jon Bentley's reversal trick [1, p. 14].

```
cilk rol(A, n, k) {
  spawn reverse(a, 0, k);
  spawn reverse(a, k, n);
  sync;
  spawn reverse(a, 0, n);
}
```

Since this complexity analysis is restricted to matrices which dimensions are powers of two, the work of `reverse` only needs to be known for arrays of even length.

Lemma 3 (Work of `reverse`). *If length $n = m_1 - m_0$ of array A is even, then*

$$W_1^{reverse}(n) = n - 1$$

Proof. The work consists of inner nodes in call-trees and swaps; we first count function calls and then the number of swaps.

`reverse` is called with recursion parameter $l = n/2$ and has base case $l = 1$. That means the binary tree that is spawned by this recursive algorithm has $n/2 - 1$ inner nodes, which corresponds to a work of $n/2 - 1$ that is required for function calls.

Each leaf of the call tree calls 'swap' once, which amounts to one unit of work per leaf, i.e. $n/2$ units of work in total, for swapping.

Summing up work for tree spawning (function calls) and swapping results in work

$$W_1^{\text{reverse}}(n) = \frac{n}{2} - 1 + \frac{n}{2} = n - 1$$

□

Work of `rol` is calculated for half-rotations. This is because array rotations in `TRIP` depend on the number of rows/columns p and q of the sub-matrices, into which a rectangular matrix is divided. Condition 5.1 results in those divisions being symmetrical, i.e. $p = q$ in this complexity analysis.

Lemma 4 (Work of `rol`)**.** *If length n of array A is even, then the work of $rol(n, n/2)$ is*

$$W_1^{rol}(n, n/2) = 2n - 3$$

Proof. `rol` consists of three reversals of lengths $n/2$, $n/2$ and n.

According to Lemma 3 the first two reversals contribute $n/2 - 1$ units of work each, and the last reversal contributes $n - 1$ units of work.

Since `rol` is not a recursive algorithm, there is no call tree whose inner nodes need to be counted. Summing up all work contributions results in

$$W_1^{\text{rol}}(n, n/2) = 2\left(\frac{n}{2} - 1\right) + (n - 1) = 2n - 3$$

□

Both Lemmas will be used when deriving the work complexity of `TRIP`.

Work of `TRIP`

Theorem 2 will derive the work of `TRIP` under constraint 5.1. Since `TRIP`, `merge` and `split` are symmetrical w.r.t. the two dimensions of a matrix save for the order of the operations, it suffices to calculate the work of calling `TRIP` on a tall matrix. The work of transposing $A_{M,N}$ is equal to the work of transposing $A_{N,M}$.

The total work consists of

1. spanning the divide tree
2. combining the nodes via `merge`/`split` (itself recursive procedures)
3. square-transposing in the leaf nodes

Theorem 2 (Work of `TRIP` for tall matrices)**.** *Let $A_{M,N}$ be a **tall** matrix that satisfies Condition 5.1.*

Then

$$W_1^{TRIP}(M, N) = \Theta\left(MN\left(1 + \log\frac{M}{N}\log N\right)\right)$$

Proof. Begin by counting the number of inner nodes in the divide tree.

Spanning the divide tree First consider the number of inner nodes in the call tree.

The recursion parameter of transpose is m, starting with $m_0 = M$, the number of rows in the matrix (or sub-matrix). From the definition of `TRIP` and Condition 5.1, it follows that

$$\forall_{0 \le i < \log\frac{M}{N}} m_{i+1} = m_i/2$$

The base case of the transpose recursion is $m = N$, which means the sub-matrix is square. As a binary tree, the transpose-tree has $\log\frac{M}{N}$ levels, $\frac{M}{N}$ leaves, and $\frac{M}{N} - 1$ inner nodes.

Combining the nodes via `merge` The number of inner nodes at level i is 2^i. So at each level `merge` is called 2^i times. The parameterization of `merge` is different for each level.

At level 0 merge($m_0/2, m_0/2, N$) is called. At level 1 merge($m_1/2, m_1/2, N$) is called, etc.

In general, at level i merge($m_i/2, m_i/2, N$) is called. Since $m_0 = M$, in general $m_i/2 = M2^{-(i+1)}$ which results in the following equation for the work of `TRIP`:

$$W_1^{\mathtt{TRIP}}(M,N) = \underbrace{\frac{M}{N} - 1}_{\text{\# of inner nodes}} + \underbrace{\sum_{0 \le i < \log \frac{M}{N}} 2^i W_1^{\text{merge}}\left(p_i, q_i, N\right)}_{\text{combine effort}} + \underbrace{\frac{M}{N} W_1^{\text{square}}(N)}_{\text{work in leaves of transpose tree (square_transpose)}} \tag{5.2}$$

So the combine effort at level i of the `TRIP` tree are 2^i `merge` calls with parameters $p_i = q_i = m_i/2 = M2^{-(i+1)}$ and recursion parameter n with $n_0 = N$. The base case of `merge` is met when $n_j = 1$ for some j. Condition 5.1 ensures that N is a power of 2, and hence repeatably divisible by two without remainder. Consequently, the binary call-tree of every `merge` call has $\log N$ levels and exactly $N - 1$ inner nodes. Contrary to the transpose tree, the base case of a merge tree does not cause work, all work is done in inner nodes. The recursion parameter n, as hinted above, takes the values $n_0 = N$, $n_1 = n_0/2$, $n_2 = n_1/2$ and so on. In general $n_{j+1} = n_j/2$, i.e. $n_j = N2^{-j}$ for $0 \le j < \log N$. Given the symmetry of the merge tree (which is based upon Condition 5.1) there are 2^j inner nodes at level j of any merge tree.

Summing up the results within the previous paragraph

$$W_1^{\text{merge}}\left(p_i, q_i, N\right) = \underbrace{N - 1}_{\text{\# of inner nodes}} + \underbrace{\sum_{0 \le j < \log N} 2^j W_1^{\text{rol}}\left(\frac{n_j}{2}p_i + \frac{n_j}{2}q_i, \frac{n_j}{2}p_i\right)}_{\text{work within inner nodes of the merge tree}} \tag{5.3}$$

Since $p_i = q_i = M2^{-(i+1)}$ and $n_j = N2^{-j}$

$$\begin{aligned}
W_1^{\text{rol}}\left(\frac{n_j}{2}p_i + \frac{n_j}{2}q_i, \frac{n_j}{2}p_i\right) &= W_1^{\text{rol}}\left(\frac{n_j}{2}(p_i + q_i), \frac{n_j}{2}p_i\right) \\
&= W_1^{\text{rol}}\left(\frac{n_j}{2}\left(M2^{-(i+1)} + M2^{-(i+1)}\right), \frac{n_j}{2}M2^{-(i+1)}\right) \\
&= W_1^{\text{rol}}\left(n_j M2^{-(i+1)}, \frac{n_j}{2}M2^{-(i+1)}\right) \\
&= W_1^{\text{rol}}\left(N2^{-j}M2^{-(i+1)}, \frac{N2^{-j}}{2}M2^{-(i+1)}\right) \\
&= W_1^{\text{rol}}\left(NM2^{-i-j-1}, \frac{1}{2}NM2^{-i-j-1}\right)
\end{aligned}$$

Lemma 4 states for even n:

$$W_1^{\mathrm{rol}}(n, n/2) = 2n - 3$$

$NM2^{-i-j-1}$ is even, so

$$\begin{aligned} W_1^{\mathrm{rol}}\left(\frac{n_j}{2}p_i + \frac{n_j}{2}q_i, \frac{n_j}{2}p_i\right) &= W_1^{\mathrm{rol}}\left(NM2^{-i-j-1}, \frac{1}{2}NM2^{-i-j-1}\right) \\ &= NM2^{-i-j} - 3 \end{aligned}$$

Plug this in Equation 5.3 to get

$$\begin{aligned} W_1^{\mathrm{merge}}(p_i, q_i, N) &= N - 1 + \sum_{0 \le j < \log N} 2^j W_1\left(\mathrm{rol}\left(\frac{n_j}{2}p_i + \frac{n_j}{2}q_i, \frac{n_j}{2}p_i\right)\right) \\ &= N - 1 + \sum_{0 \le j < \log N} 2^j \left(NM2^{-i-j} - 3\right) \\ &= N - 1 + \sum_{0 \le j < \log N} \left(NM2^{-i} - 3 \cdot 2^j\right) \\ &= N - 1 + \log(N)NM2^{-i} - 3 \sum_{0 \le j < \log N} 2^j \\ &= N - 1 + \log(N)NM2^{-i} - 3(N - 1) \\ &= \log(N)NM2^{-i} - 2(N - 1) \end{aligned}$$

Plugging this in Equation 5.2 results in

$$\begin{aligned} W_1^{\mathtt{TRIP}}(M, N) &= \left(\frac{M}{N} - 1\right) + \sum_{0 \le i < \log \frac{M}{N}} 2^i W_1^{\mathrm{merge}}(p_i, q_i, N) + \frac{M}{N} W_1^{\mathrm{square}}(N) \qquad (5.4) \\ &= \left(\frac{M}{N} - 1\right) + \sum_{0 \le i < \log \frac{M}{N}} 2^i \left(\log(N)NM2^{-i} - 2(N - 1)\right) + \frac{M}{N} W_1^{\mathrm{square}}(N) \\ &= \left(\frac{M}{N} - 1\right) + \left(\log\frac{M}{N} \log(N)NM - 2(N - 1)\left(\frac{M}{N} - 1\right)\right) + \frac{M}{N} W_1^{\mathrm{square}}(N) \\ &= (-2N + 3)\left(\frac{M}{N} - 1\right) + \left(MN \log\frac{M}{N} \log N\right) + \frac{M}{N} W_1^{\mathrm{square}}(N) \end{aligned}$$

Work of square transpose The work of `square_transpose`(N) consists of spanning the recursion-tree and swapping the non-diagonal entries. The number of spanned sub-trees in each inner node of the recursion depends on the position of the associated sub-matrix within the square matrix and is either three or four. This paragraph contains a derivation of an asymptotically tight upper and lower bound for the work complexity of `square_transpose`.

First asymptotically determine the number of inner nodes of the tree. The tree spans the upper right triangular part of the matrix, including the diagonal, so there are $N(N + 1)/2$ leaves. The following will show that for any arity of the tree, be it three or four, the number of inner nodes of said tree is $\Theta(N^2)$.

A ternary tree with $N(N+1)/2$ leaves has $\lceil \log_3(N(N+1)/2) \rceil$ levels. The number of inner nodes of such a tree is

$$\sum_{i=0}^{\lceil \log_3(N(N+1)/2)-1 \rceil} 3^i$$

For big N approximate $\lceil \log_3(N(N+1)/2) - 1 \rceil$ by $\log_3(N(N+1)/2) - 1$. Since

$$\sum_{k=0}^{m} 3^i = \frac{1}{2}\left(3^{m+1} - 1\right)$$

the number of inner nodes of the tree is

$$\sum_{i=0}^{\log_3(N(N+1)/2)-1} 3^i = \frac{1}{2}\left(3^{\log_3(N(N+1)/2)} - 1\right) = \frac{N(N+1)}{4} - \frac{1}{2} = \Theta(N^2)$$

A quaternary tree with $N(N+1)/2$ leaves has $\lceil \log_4(N(N+1)/2) \rceil$ levels. The number of inner nodes of such a tree is

$$\sum_{i=0}^{\lceil \log_4(N(N+1)/2)-1 \rceil} 4^i$$

For big N approximate $\lceil \log_4(N(N+1)/2) - 1 \rceil$ by $\log_4(N(N+1)/2) - 1$. Since

$$\sum_{k=0}^{m} 4^i = \frac{1}{3}\left(4^{m+1} - 1\right)$$

the number of inner nodes of the tree is

$$\sum_{i=0}^{\log_4(N(N+1)/2)-1} 4^i = \frac{1}{3}\left(4^{\log_4(N(N+1)/2)} - 1\right) = \frac{N(N+1)}{6} - \frac{1}{3} = \Theta(N^2)$$

In any combination of cases that are matched during the recursion of `square_transpose` the work of spanning the tree is $\Theta(N^2)$.

The work of swapping the elements occurs in the leaves of the tree. All elements above but excluding the diagonal need to be swapped with elements below the diagonal, i.e. $N(N-1)/2$ swaps transpose the matrix.

In total the work for spanning the transpose tree and swapping is

$$W_1^{\text{square}}(N) = \Theta(N^2) + N(N-1)/2 = \Theta(N^2)$$

Plugged into equation 5.4 this results in

$$\begin{aligned} W_1^{\texttt{TRIP}}(M,N) &= (-2N+3)\left(\frac{M}{N} - 1\right) + \left(MN \log \frac{M}{N} \log N\right) + \frac{M}{N} W_1^{\text{square}}(N) \\ &= (-2N+3)\left(\frac{M}{N} - 1\right) + \left(MN \log \frac{M}{N} \log N\right) + \frac{M}{N}\Theta(N^2) \end{aligned} \tag{5.5}$$

This simplifies to

$$
\begin{aligned}
W_1^{\texttt{TRIP}}(M,N) &= \Theta\left(M + MN\log\frac{M}{N}\log N + MN\right)\\
&= \Theta\left(MN\left(1+\log\frac{M}{N}\log N\right)\right)
\end{aligned}
$$

□

This result can be generalized, to also holds for wide matrices:

Corollary 1 (Work of `TRIP`)**.** *Let $A_{M,N}$ be a tall, wide or square matrix that satisfies Condition 5.1. Then*

$$
W_1^{TRIP}(M,N) = \Theta\left(MN\left(1+\log\frac{\max(M,N)}{\min(M,N)}\log\min(M,N)\right)\right)
$$

Proof. The `TRIP` recursion is analogous for tall and wide matrices, furthermore `merge` and `split` call `rol` with the same arguments (under Constraint 5.1 $p = q$ and $\lceil\frac{m}{2}\rceil = \lfloor\frac{m}{2}\rfloor$ on the respective levels in the `merge`/`split`-tree). The only difference between applying `TRIP` to a tall or a wide matrix occurs inside `merge` and `split`: In `merge` `rol` is called before the recursive `merge` call, in `split` `rol` is called after the recursive call. This difference does not cause a change in the amount of work of `TRIP`. □

The next section contains the derivation of the asymptotic span of `TRIP`.

5.2 Span

In this section, after a brief definition of the span of a computation, Lemmas 5 and 6 cover the span of the algorithms `reverse` and `rol`. After that, Theorem 3 shows, that the transpose of a tall matrix $A_{M,N}$ has span

$$
W_\infty^{\texttt{TRIP}}(M,N) = \Theta\left(\log\frac{M}{N}\log^2 N + \log^2\frac{M}{N}\log N\right)
$$

provided M and N fulfill Condition 5.1. Corollary 2 generalizes this result to wide and square matrices.

In the context of the dag model described in [5, p.777ff], the ***span*** W_∞ is defined as the longest path in a computation dag. In practical terms it can be interpreted as the number of instructions of an algorithm that need to be executed in serial, even if infinitely many processors were available. This analysis does not count single instructions, but instruction-groups. Like in the analysis of W_1 in the last chapter, we consider as instruction groups

- inner nodes in a spawned tree structure, and
- swaps.

Each level in a recursive call tree, which most of the time amounts to about three actual function calls, is considered one node in the critical path of the computation dag, as is a swap of two array entries.

The span analysis of two algorithms that form the base of `TRIP` is a good introduction to the general principles applied throughout this section.

Span of Base Algorithms

The span complexity of `reverse` and `rol` depends on the applied algorithms. This analysis is based on the algorithms `reverse` and `rol`, that are listed in Section 5.1.

Since this complexity analysis is restricted to matrices which dimensions are powers of two, the span of `reverse` only needs to be known for arrays of even length.

Lemma 5 (Span of `reverse`)**.** *If the length n of array A is a power of two, then*

$$W_\infty^{reverse}(n) = \log \frac{n}{2} + 1$$

Proof. The span consists of the number of subsequent operations in the recursion tree (the depth of the tree), and the span of the base case. In each recursive call, l is halved. Since initially $l = (m_1 - m_0)/2 = n/2$, this results in a tree depth of $\log n/2$. In the base case, one element of the array is swapped, which adds 1 to the span $W_\infty^{\texttt{reverse}}$.

Summing up the span of tree spawning and swapping results in

$$W_\infty^{\texttt{reverse}}(n) = \log \frac{n}{2} + 1$$

□

Span of `rol` is calculated for half-rotations. This is because array rotations in `TRIP` depend on the number of rows/columns p and q of the sub-matrices, into which a rectangular matrix is divided. Condition 5.1 results in those divisions being symmetrical, i.e. $p = q$ in this complexity analysis.

Lemma 6 (Span of `rol`)**.** *If length n of array A is a power of two, then the span of $rol(n, n/2)$ is*

$$W_\infty^{rol}(n) = \log 2^{-1}n + \log 2^{-2}n + 3$$

Proof. `rol` consists of three reversals of lengths $n/2$, $n/2$ and n, where the first two reversals are executed in parallel.

According to Lemma 3 the first two reversals have span $n/4 + 1$. Since they are executed in parallel, and the longest paths of either branch are equally long, the combined span of both reversals is $n/4 + 1$. The last reversal contributes span $n/2 + 1$, since it is executed in serial to the first two reversals.

Summing up all contributions and adding span 1 for the high-level function calls results in

$$W_\infty^{\text{rol}}(n, n/2) = \log \frac{n}{2} + \log \frac{n}{4} + 3 = \log 2^{-1}n + \log 2^{-2}n + 3$$

□

Both Lemmas will be used when deriving the span of `TRIP`.

Span of TRIP

Theorem 3 will derive the span of TRIP under constraint 5.1. Since TRIP, merge and split are symmetrical w.r.t. the two dimensions of a matrix save for the order of the operations, it suffices to calculate the span of calling TRIP on a tall matrix. The work of transposing $A_{M,N}$ is equal to the work of transposing $A_{N,M}$.

The total span consists of three contributions

1. spanning the divide tree

2. combining the nodes via merge/split (itself recursive procedures)

3. square-transposing in the leaf nodes

Theorem 3 (Span of TRIP for tall matrices)**.** *Let $A_{M,N}$ be a **tall** matrix that satisfies Condition 5.1. Then*

$$W_\infty^{TRIP}(M,N) = \Theta\left(\log \frac{M}{N} \log^2 N + \log^2 \frac{M}{N} \log N\right)$$

Proof. We begin by counting the levels in the divide tree.

Spanning the divide tree First we consider the levels in the call tree.

The recursion parameter of transpose is m, starting with $m_0 = M$, the number of rows in the matrix (or sub-matrix). From the definition of TRIP and Condition 5.1 follows

$$\forall_{0 \le i < \log \frac{M}{N}} m_{i+1} = m_i/2$$

The base case of the transpose recursion is $m = N$, which means the sub-matrix is square. As a binary tree, the transpose-tree has $\log \frac{M}{N}$ levels.

Combining the nodes via merge The parameterization of merge is different for each level.

At level 0 merge$(m_0/2, m_0/2, N)$ is called. At level 1 merge$(m_1/2, m_1/2, N)$ is called, etc.

This is done $\log M/N$ times, for $0 \le i < \log M/N$, until $m_{\log M/N-1} = N$. In general, at level i $m_i = N2^{\log M/N-i}$, and merge$(m_i/2, m_i/2, N)$ is called.

This results in the following equation for the span of TRIP:

$$W_\infty^{\texttt{TRIP}}(M,N) = \underbrace{\log \frac{M}{N}}_{\text{levels in TRIP tree}} + \underbrace{\sum_{0 \le i < \log \frac{M}{N}} W_\infty^{\texttt{merge}}(p_i, q_i, N)}_{\text{levels in merge trees}} + \underbrace{W_\infty^{\text{square}}(N)}_{\text{span of leaf of transpose tree (square_transpose)}} \tag{5.6}$$

Continue by calculating the span of a merge call at level i in the transpose tree.

The span of combining the recursive `TRIP` calls with `merge` at level i of the `TRIP` tree is the span of a `merge` call with parameters $p_i = q_i = m_i/2 = N2^{\log M/N-i+1}$ and recursion parameter $n_0 = N$. The base case of `merge` is met when $n_j = 1$ for some j. Condition 5.1 ensures that N is a power of 2, and hence repeatably divisible by two without remainder. Consequently, The binary call-tree of every `merge` call has $\log N$ levels. Contrary to the transpose tree, the base case of a merge tree does not increase span, since all work is done in inner nodes.

Summing up the results within the previous paragraph

$$W_\infty^{\texttt{merge}}\left(p_i, q_i, N\right) = \underbrace{\log N}_{\text{\# of levels of merge tree}} + \underbrace{\sum_{0\le j<\log N} W_\infty^{\texttt{rol}}\left(\frac{n_j}{2}p_i + \frac{n_j}{2}q_i, \frac{n_j}{2}p_i\right)}_{\text{span of inner nodes of the merge tree}} \tag{5.7}$$

Since $p_i = q_i = N2^{\log M/N-i-1}$ and $n_j = 2^{\log N-j}$

$$\begin{aligned}
W_\infty^{\texttt{rol}}\left(\frac{n_j}{2}p_i + \frac{n_j}{2}q_i, \frac{n_j}{2}p_i\right) &= W_\infty^{\texttt{rol}}\left(\frac{n_j}{2}(p_i+q_i), \frac{n_j}{2}p_i\right) \\
&= W_\infty^{\texttt{rol}}\left(\frac{n_j}{2}\left(2N2^{\log M/N-i-1}\right), \frac{n_j}{2}N2^{\log M/N-i-1}\right) \\
&= W_\infty^{\texttt{rol}}\left(n_j N2^{\log M/N-i-1}, \frac{n_j}{2}N2^{\log M/N-i-1}\right) \\
&= W_\infty^{\texttt{rol}}\left(2^{\log N-j}N2^{\log M/N-i-1}, \frac{2^{\log N-j}}{2}N2^{\log M/N-i-1}\right) \\
&= W_\infty^{\texttt{rol}}\left(N2^{\log N+\log M/N-i-j-1}, N2^{\log N+\log M/N-i-j-2}\right)
\end{aligned}$$

Lemma 6 states for n that are powers of two:

$$W_\infty^{\texttt{rol}}(n, n/2) = \log 2^{-1}n + \log 2^{-2}n + 3$$

$N2^{\log N+\log M/N-i-j-1}$ is a power of two, so

$$\begin{aligned}
&W_\infty^{\texttt{rol}}\left(N2^{\log N+\log M/N-i-j-1}, N2^{\log N+\log M/N-i-j-2}\right) = \\
&= \log\left(N2^{\log N+\log M/N-i-j-2}\right) + \log\left(N2^{\log N+\log M/N-i-j-3}\right) + 3 \\
&= 2\log N + 2(\log N + \log M/N - i - j) - 2 \\
&= 4\log N + 2\log M/N - 2i - 2j - 2
\end{aligned}$$

Plug this span of `rol` into equation 5.7 and simplify to get $W_\infty^{\texttt{merge}}$:

$$\begin{aligned}
W_\infty^{\texttt{merge}}\left(p_i, q_i, N\right) &= \log N + \sum_{0\le j<\log N} W_\infty^{\texttt{rol}}\left(\frac{n_j}{2}p_i + \frac{n_j}{2}q_i, \frac{n_j}{2}p_i\right) \\
&= \log N + \sum_{0\le j<\log N} (4\log N + 2\log M/N - 2i - 2j - 2) \\
&= \log N\left(4\log N + 2\log\frac{M}{N} - 2i - 2\right) - 2\sum_{0\le j<\log N} j \\
&= \log N\left(4\log N + 2\log\frac{M}{N} - 2i - 2\right) - \left(\log^2 N - \log N\right) \\
&= 3\log^2 N + 2\log\frac{M}{N}\log N - 2i\log N - 1\log N
\end{aligned}$$

Plug this result into equation 5.6 to derive the span of `TRIP` excluding the square transpose in the base case:

$$\begin{aligned}
W_\infty^{\texttt{TRIP}}(M, N) &= \log\frac{M}{N} + \sum_{0\le i<\log\frac{M}{N}} W_\infty^{\texttt{merge}}\left(p_i, q_i, N\right) + W_\infty^{\text{square}}(N) \qquad (5.8)\\
&= \log\frac{M}{N} + \sum_{0\le i<\log\frac{M}{N}}\left(3\log^2 N + 2\log\frac{M}{N}\log N - 2i\log N - 1\log N\right) + W_\infty^{\text{square}}(N) \\
&= \log\frac{M}{N} + \log\frac{M}{N}\left(3\log^2 N + 2\log\frac{M}{N}\log N - 1\log N\right) \\
&\qquad - \log N\left(\log^2\frac{M}{N} - \log\frac{M}{N}\right) + W_\infty^{\text{square}}(N) \\
&= \log\frac{M}{N} + \log\frac{M}{N}\log N\left(3\log N + \log\frac{M}{N}\right) + W_\infty^{\text{square}}(N)
\end{aligned}$$

The proof is finished by calculating the span of the square transpose.

Span of square transpose The work of `square_transpose`(N) consists of spanning the recursion-tree and one swap in the base case of the recursion. The depth of the recursion is $\log N$ because the tree is spawned two-dimensionally, and each recursive call halves the number of rows as well as the number of columns in the arguments of the recursive call.

Using

$$W_\infty^{\text{square}}(N) = \log N + 1$$

We conclude the proof by applying W_∞^{square} in 5.8 to get

$$\begin{aligned}
W_\infty^{\texttt{TRIP}}(M, N) &= \log\frac{M}{N} + \log\frac{M}{N}\log N\left(3\log N + \log\frac{M}{N}\right) + \log N + 1 \\
&= \Theta\left(\log\frac{M}{N}\log^2 N + \log^2\frac{M}{N}\log N\right)
\end{aligned}$$

□

This result can be generalized, to also holds for wide matrices:

Corollary 2 (Span of TRIP). *Let $A_{M,N}$ be a tall, wide or square matrix that satisfies Condition 5.1. Then*

$$W_{\infty}^{TRIP}(M, N) = \Theta\left(\log \frac{m}{n} \log^2 n + \log^2 \frac{m}{n} \log n\right)$$

where $m = \max(M, N)$ *and* $n = \min(M, N)$.

Proof. The TRIP recursion is analogous for tall and wide matrices, furthermore merge and split call rol with the same arguments (under Constraint 5.1 $p = q$ and $\lceil \frac{n}{2} \rceil = \lfloor \frac{m}{2} \rfloor$ on the respective levels in the merge/split-tree). The only difference between applying TRIP to a tall or a wide matrix occurs inside merge and split: In merge rol is called before the recursive merge call, in split rol is called after the recursive call. This difference does not cause a change in the span of TRIP. □

5.3 Parallelism

This section covers the derivation of parallelism of $\Theta\left(\frac{MN}{\log M/N + \log N}\right)$ for rectangular matrices $A_{M,N}$ that fulfill Condition 5.1, and parallelism $\Theta\left(\frac{N^2}{\log N}\right)$ for square matrices.

The derivation consists of dividing work by span, a case distinction between rectangular and square matrices, and simplification using Landau symbols.

Recall work and span for $M \geq N$:

$$W_1^{\text{TRIP}}(M, N) = \Theta\left(MN\left(1 + \log \frac{M}{N} \log N\right)\right)$$

$$W_{\infty}^{\text{TRIP}}(M, N) = \log \frac{M}{N} + \log \frac{M}{N} \log N \left(3 \log N + \log \frac{M}{N}\right) + \log N + 1$$

Distinguish between the cases $M = N$ and $M > N$.

Case 1: $M = N$ In this case work and span simplify to

$$W_1^{\text{TRIP}}(N, N) = \Theta\left(N^2\right)$$

and

$$W_{\infty}^{\text{TRIP}}(M, N) = \Theta(\log N)$$

This results in parallelism

$$\frac{W_1^{\text{TRIP}}}{W_{\infty}^{\text{TRIP}}}(N, N) = \Theta\left(\frac{N^2}{\log N}\right)$$

Next, consider the rectangular case

Case 2: $M > N$ In this case work and span simplify too, because $\log N \geq 1$ and $\log \frac{M}{N} \geq 1$:

$$W_1^{\texttt{TRIP}}(M,N) = \Theta\left(MN\left(\log\frac{M}{N}\log N\right)\right)$$

$$W_\infty^{\texttt{TRIP}}(M,N) = \Theta\left(\log\frac{M}{N}\log N\left(\log N + \log\frac{M}{N}\right)\right)$$

Dividing work by span results in parallelism

$$\frac{W_1^{\texttt{TRIP}}}{W_\infty^{\texttt{TRIP}}}(N,N) = \Theta\left(\frac{MN}{\log\frac{M}{N} + \log N}\right)$$

The next chapter contains visualizations and interpretations of the results.

5.4 Interpretation of Results

When plotted as a function of the aspect ratio of $A_{M,N}$, both work and span have the shape of "eagle curves". This section contains an interpretation of these curves.

In the following we present plots of work, span and parallelism for rectangular matrices of equal number of elements, but different aspect ratio, and optimizations.

5.4.1 Work and Span

Plotting different aspect ratios of matrices works as follows. The abscissa i parameterizes the matrices aspect ratio. For given i the matrix is of dimensions $M \times N$ with $M = 2^i$ and $N = 2^{20-i}$. The minimum in both plots is the case where $M = N$, i.e. the square case. Left of the minimum are wide matrices, with the leftmost matrix being of dimension 2×20^{19}. Right of the minimum are tall matrices, with the rightmost being the tallest. All matrices in one plot have the same number of elements.

Separation of concerns shows that the eagle shape of Figures 5.1 and Figure 5.2 is the result of two opposing trends: For slightly rectangular matrices, the trees spawned by `merge` are deep, and `merge` encounters high work and span. For very slim, highly rectangular matrices, the `TRIP` tree has more inner nodes, and the resulting amount of `merge` calls increases, though the `merge` recursion itself is less deep and encounters less work. These opposing factors multiply and result in these eagle curves.

5.4.2 Parallelism

Counted, actual values for parallelism are compared to the asymptotic results of section 5.3.

Counting work and span in an actual program execution results in exact values for parallelism. Figure 5.3 shows this comparison.

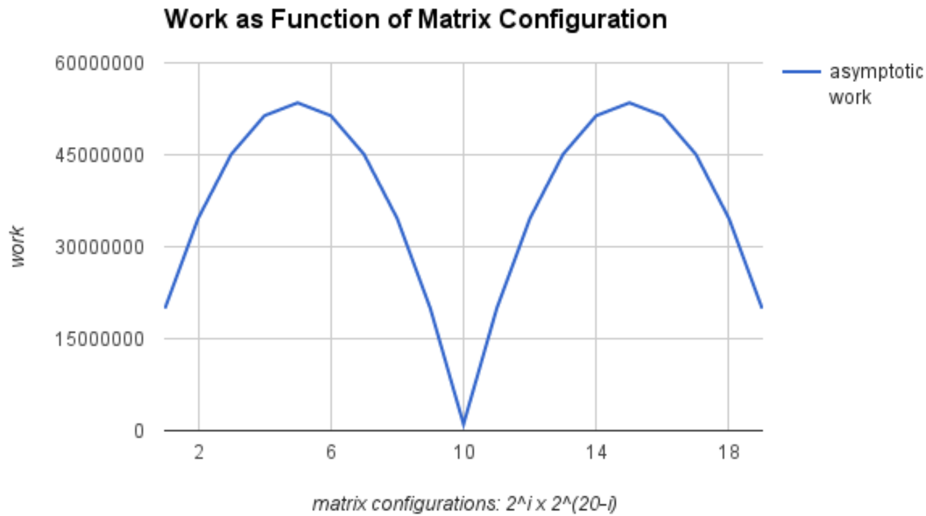


Figure 5.1: Work as function of matrix configuration (i.e. aspect ratio)

5.4.3 Analysis for Matrix Configurations that are not Powers of Two

The complexity analysis of this chapter was limited to matrices, which dimensions are powers of two. In other cases TRIP behaves less symmetrical, and is harder to analyze. This is because the sub-matrices, into which the matrix is split, are not of the same sizes in general.

This complex behavior is visible in Figure 5.4 and Figure 5.5. These figures show the counted work and span for evaluations of TRIP on matrices with about the same number of elements, allowing a 5% difference to increase the number of valid matrix configurations. In these plots the abscissa is the number of rows in the matrix, the ordinate is work and span respectively.

The analysis of Chapter 5 results in the lower hull of this curves.

5.4.4 Optimizations to the Algorithm

A simple modification that drastically changes work and span of TRIP can be made in the selection of the sub-matrices of TRIP.

The algorithm, as described in this thesis, is not optimized for matrices that are not powers of two: Fast convergence to square matrices means less merge and split calls and less work and span, but TRIP simply bisects rectangular matrices, even though the matrix might already be almost square.

The original way to split a tall matrix into sub-matrices is to divide the matrix into two sub-matrices of about half the size:

$i_m = (i_1 + i_0) / 2;$

In the case of a matrix that is almost square, this simply results in two wide sub-matrices that need to be divided again. A modification is to divide almost square matrices into one square sub-matrix and a second rectangular matrix:

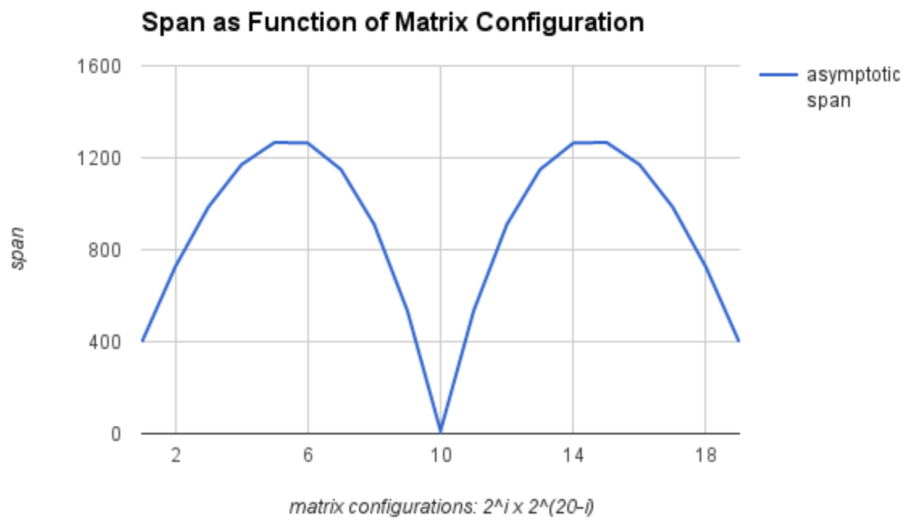


Figure 5.2: Span as function of matrix configuration (i.e. aspect ratio)

```
if (m < 2n)
  i_m = i_0 + n;
else
  i_m = (i_1 + i_0) / 2;
```

This way work can be reduced, though span cannot. This can be seen in Figure 5.6 and Figure 5.7.

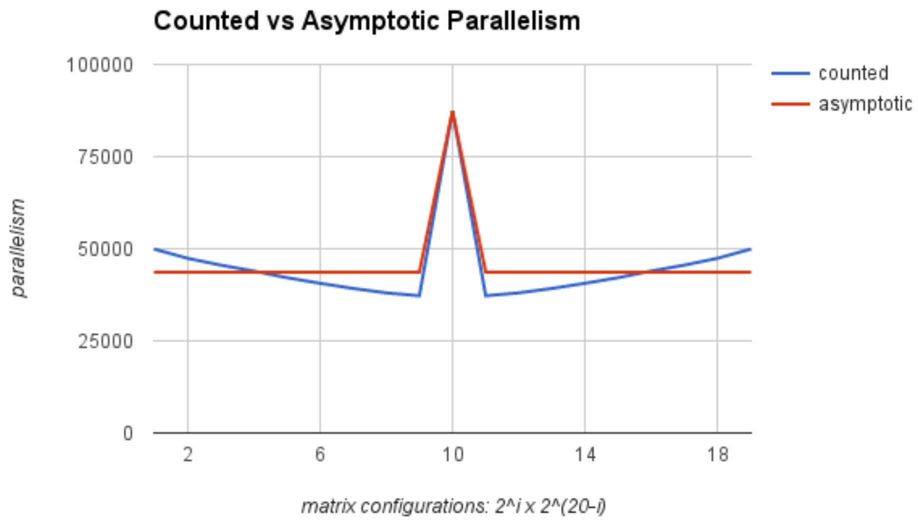


Figure 5.3: Comparison between actual parallelism and result of asymptotic analysis.

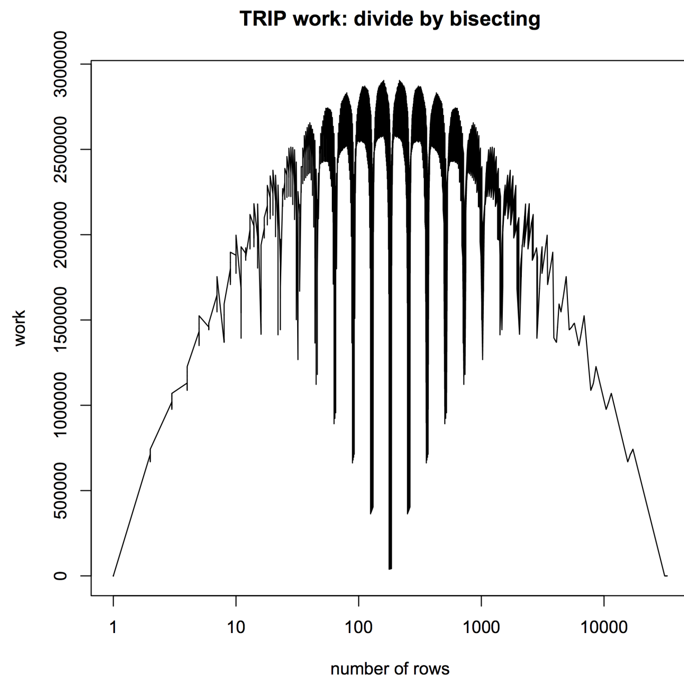


Figure 5.4: Work as function of number of rows for matrices of roughly the same number of elements (allowing a 5% difference).

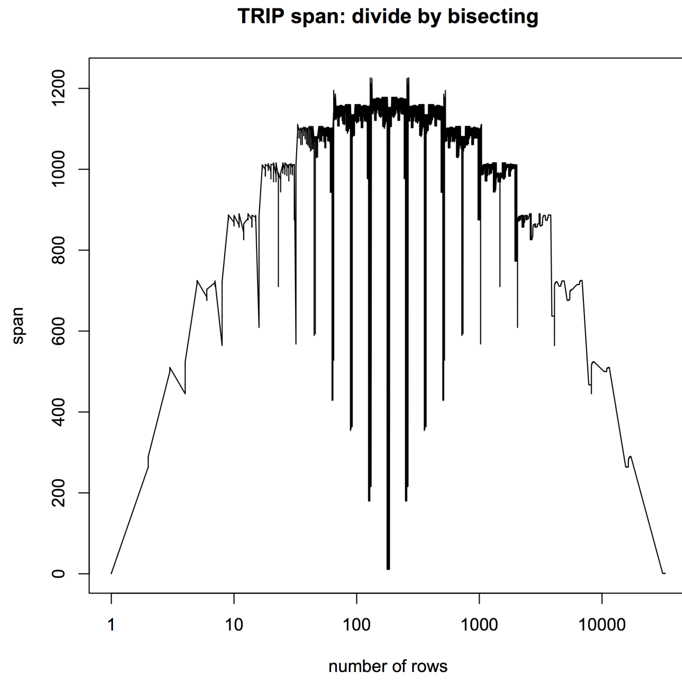


Figure 5.5: Span as function of number of rows for matrices of roughly the same number of elements (allowing a 5% difference).

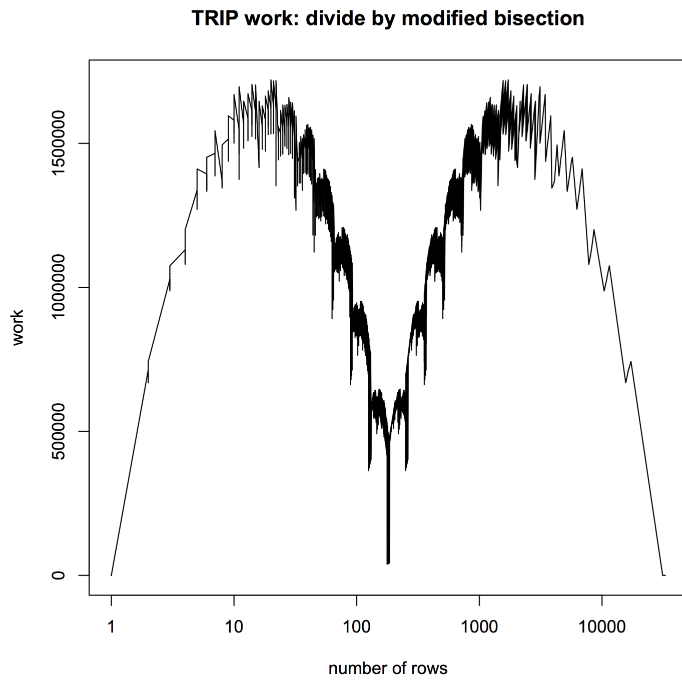


Figure 5.6: Work as function of number of rows for matrices of roughly the same number of elements (allowing a 5% difference).

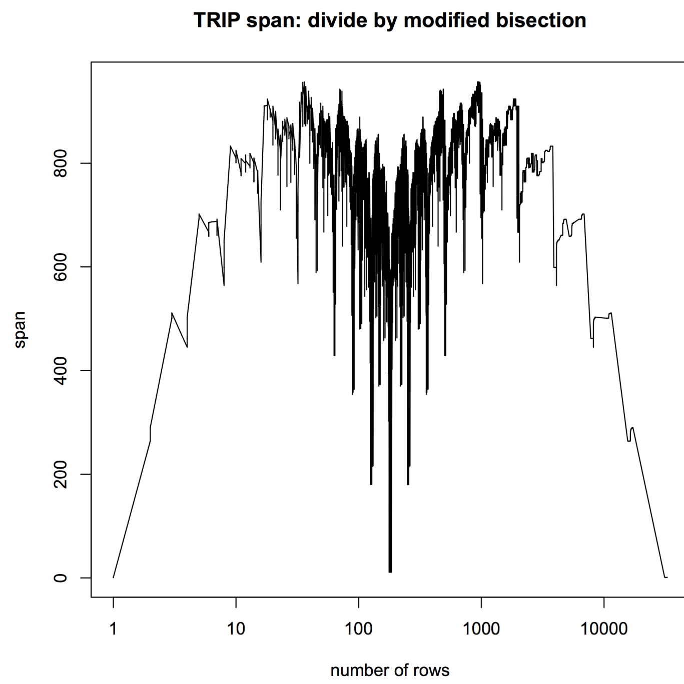


Figure 5.7: Span as function of number of rows for matrices of roughly the same number of elements (allowing a 5% difference).

6 Experimental Results

The results in this chapter are tests of performance and scalability, and underline the theoretical results of the previous chapters.

All benchmarks have been executed on the same instance of a Google Cloud Intel Haswell n1-highcpu-64 instance with 64 vCPUs, and 57.6 GB memory. All benchmarks that utilize less than 64 cores benefit from an absence of noisy neighbors, since the high core-count virtually guarantees that a whole node is allocated for the benchmark alone. This reduces noise in the benchmarks, and allows for comparable results across benchmarks.

6.1 Performance

Of work, span and parallelism, work can be benchmarked efficiently, since that requires only one CPU core, as opposed to an arbitrarily large number of cores for span and parallelism.

Work of TRIP can be analyzed in two dimensions: One is the work of matrices with changing aspect ratio and constant number of elements. This will result in eagle curves, as derived in Chapter 5.1. The second is the work of matrices with constant aspect ratio, but changing number of elements. In this part we will show the growth in computational complexity for growing matrix sizes.

6.1.1 Changing Aspect Ratio, Constant Matrix Size

Figure 6.1 shows execution time and scaled work complexity for matrices with 2^{26} elements, but varying aspect ratio. From left to right the figure shows execution times and work complexity for very wide, wide, square, tall, and very tall matrices. The chart shows the typical eagle-curve pattern and a very good agreement between timing and predicted work.

6.1.2 Changing Matrix Size, Constant Aspect Ratio

Figures 6.2, 6.3, 6.4, 6.5, and 6.6 show experiments that test the computational complexity for changing matrix size. This is done for multiple aspect ratios: for very wide, slightly wide, square, slightly tall, and tall matrices. The left vertical axis shows the execution time of TRIP for the given matrix size, the right vertical axis shows the scaled, asymptotic work complexity. All charts show close agreement between theoretical results and experiments.

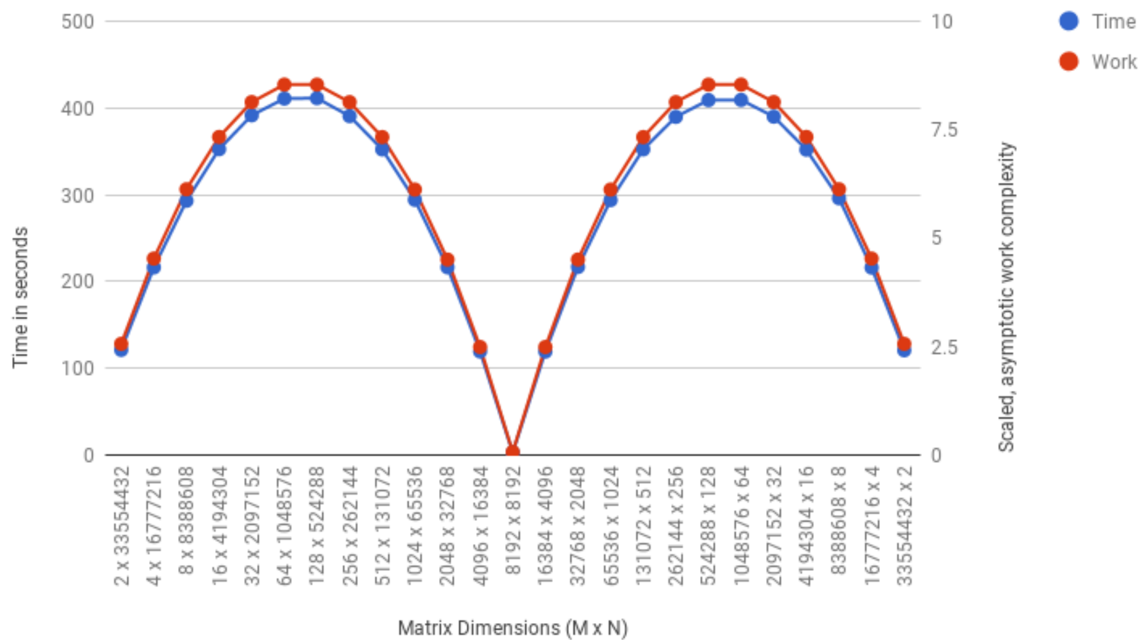


Figure 6.1: Comparison of execution time (left vertical axis) and work (right vertical axis) for matrices with 2^{26} elements and varying aspect ratio

6.2 Scalability

Scalability benchmarks test how the execution time changes, when the number of used processors increase. We depict scalability in the form of speed-up graphs: Given a fixed number of elements and a fixed configuration—again very wide, wide, square, tall, very tall—the graphs show execution time, given an increasing number of compute cores. All scalability benchmarks are executed on matrices with 2^{26} elements. That means that for a given graph, the aspect ratio as well as the size of the matrix is fixed, and only the core-count changes. The graphs in Figures 6.7, 6.8, 6.9, 6.10, and 6.11 show the speed-up w.r.t. the execution time on one core. Up until 32 cores the algorithm scales virtually perfectly; the less-than-ideal speed-up with 64 cores is suspected to occur due to hyper-threading.

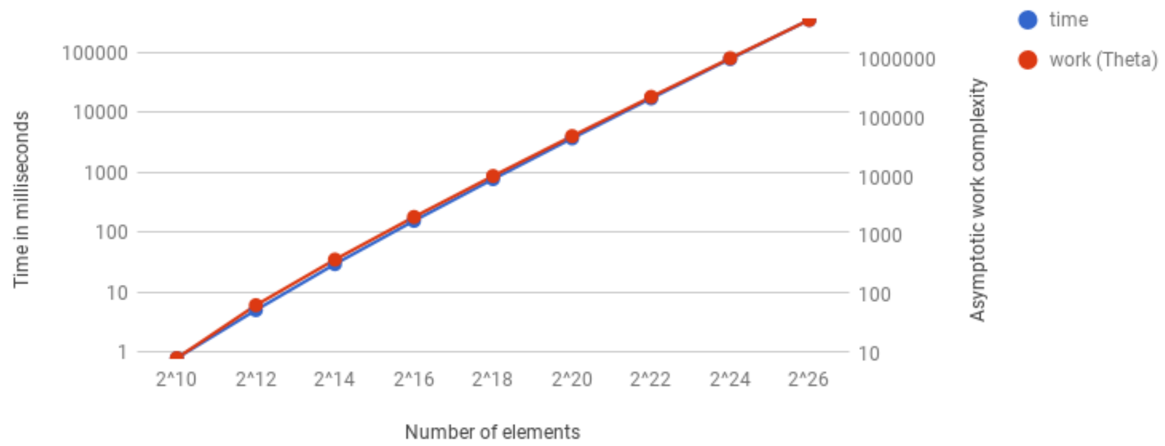


Figure 6.2: The time (left) and scaled, asymptotic work complexity (right) for very wide matrices with aspect ratio 1×256, and between 2^{10} and 2^{26} elements.

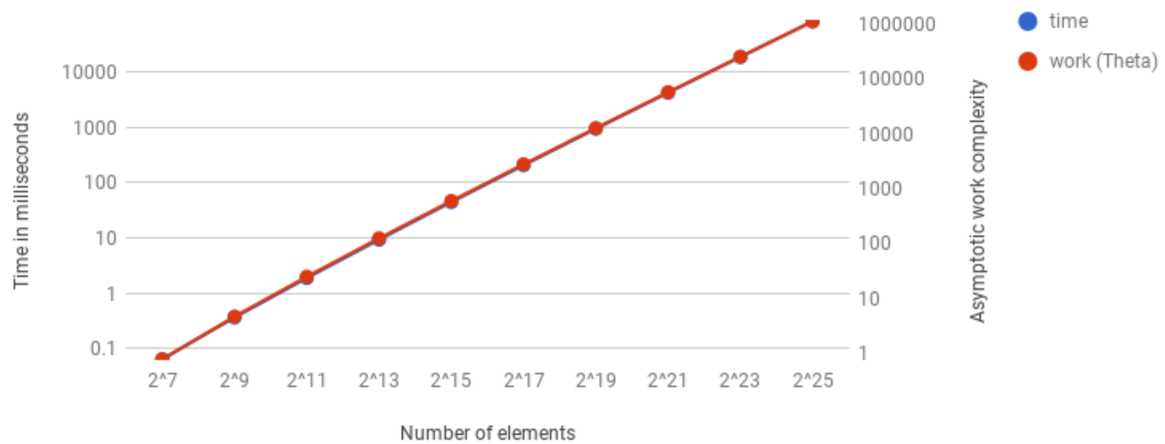


Figure 6.3: The time (left) and scaled, asymptotic work complexity (right) for wide matrices with aspect ratio 1×8, and between 2^7 and 2^{25} elements.

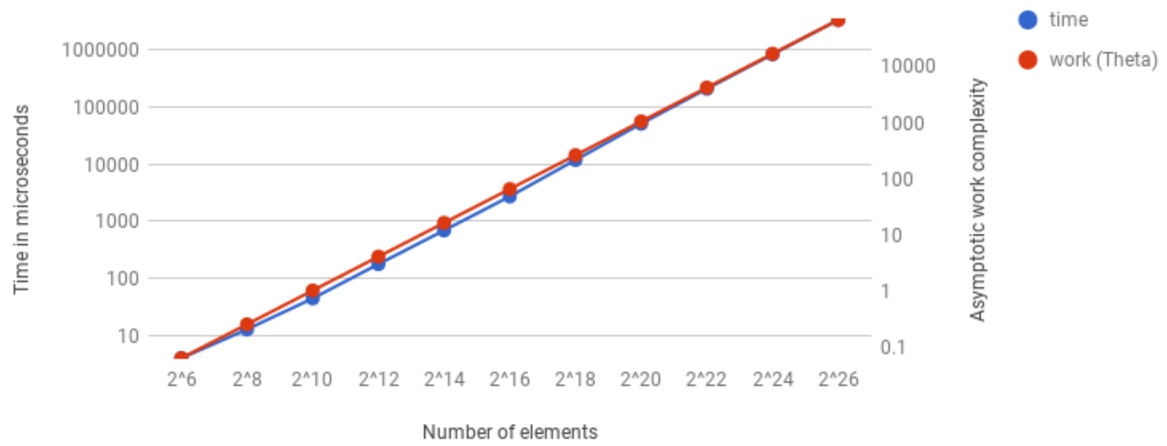


Figure 6.4: The time (left) and scaled, asymptotic work complexity (right) for square matrices, between 2^6 and 2^{26} elements.

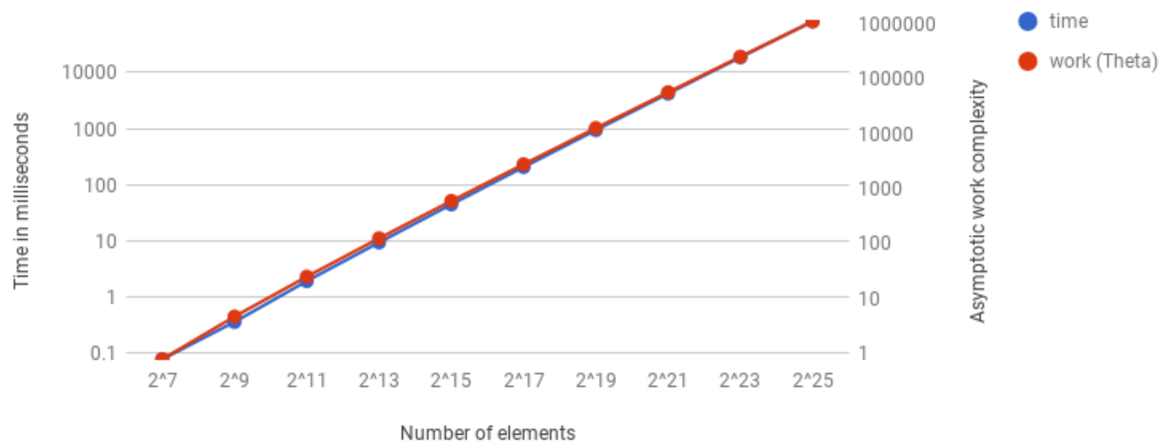


Figure 6.5: The time (left) and scaled, asymptotic work complexity (right) for tall matrices with aspect ratio 8×1, and between 2^7 and 2^{25} elements.

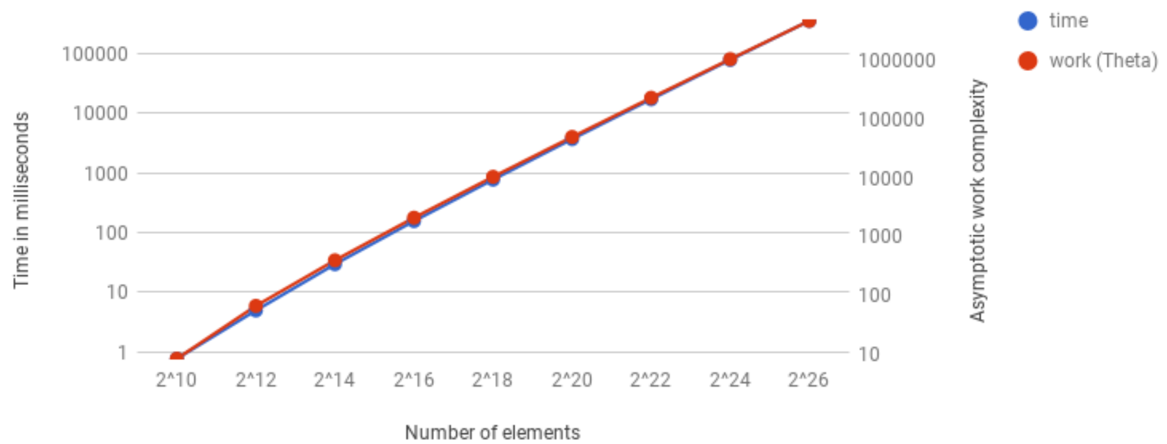


Figure 6.6: The time (left) and scaled, asymptotic work complexity (right) for very tall matrices with aspect ratio 256×1, and between 2^{10} and 2^{26} elements.

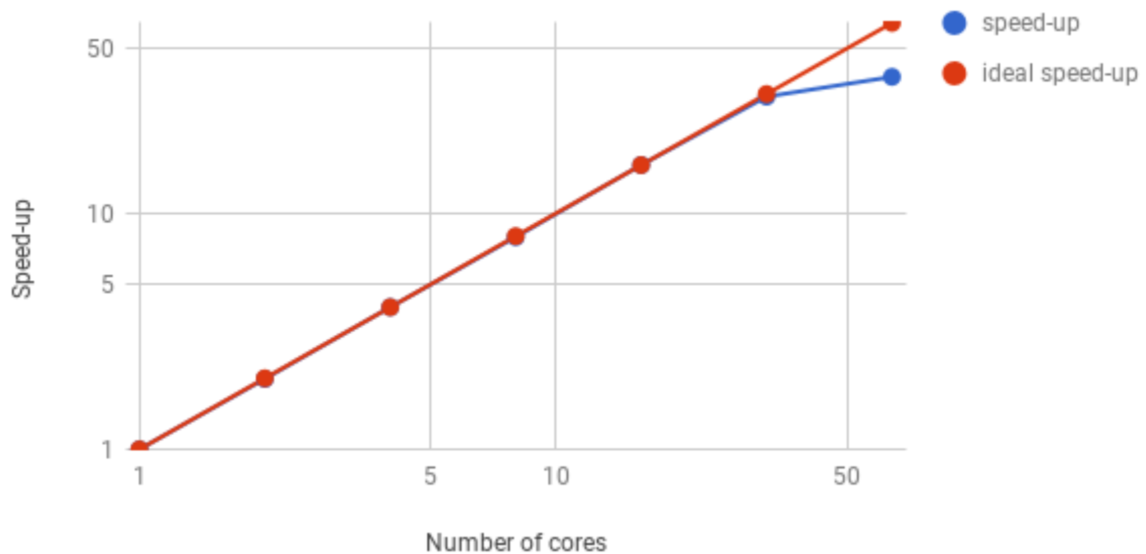


Figure 6.7: Speed-up of TRIP on a wide 2×33554432 matrix with 2^{26} entries, up to 64 cores.

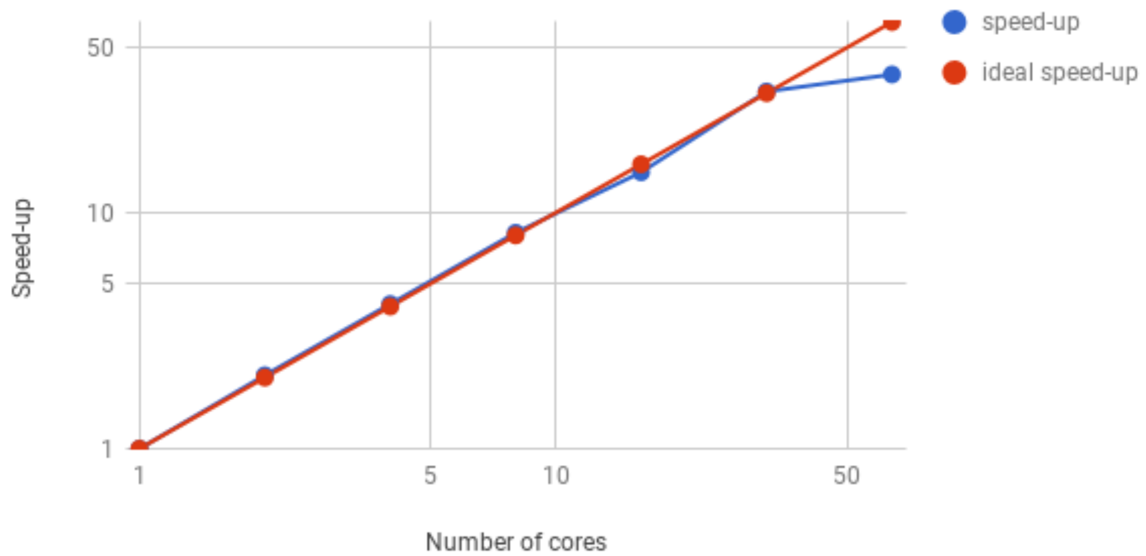


Figure 6.8: Speed-up of TRIP on a wide 512×131072 matrix with 2^{26} entries, up to 64 cores.

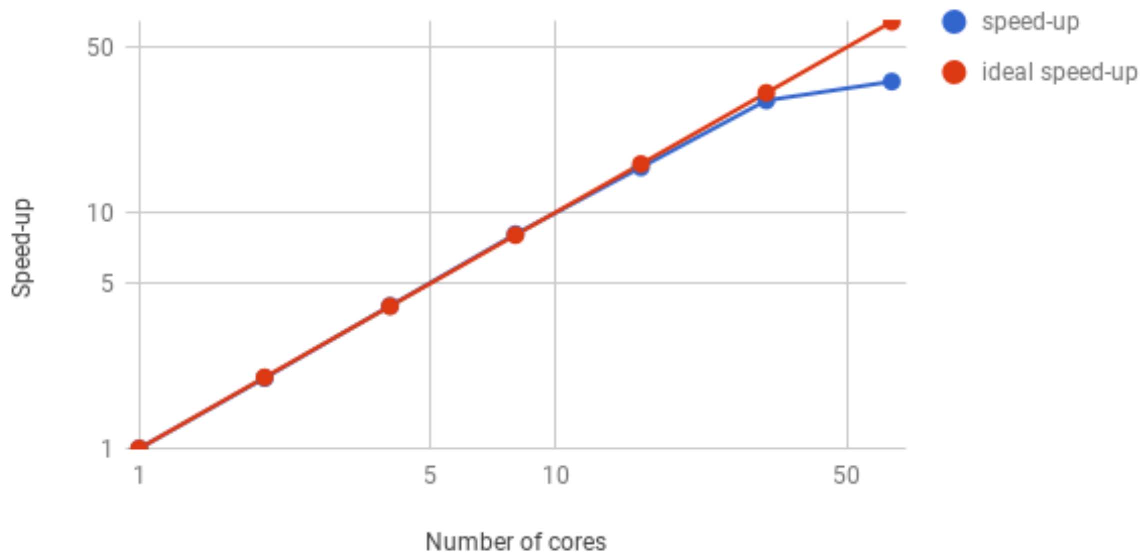


Figure 6.9: Speed-up of TRIP on a square 8192×8192 matrix with 2^{26} entries, up to 64 cores.

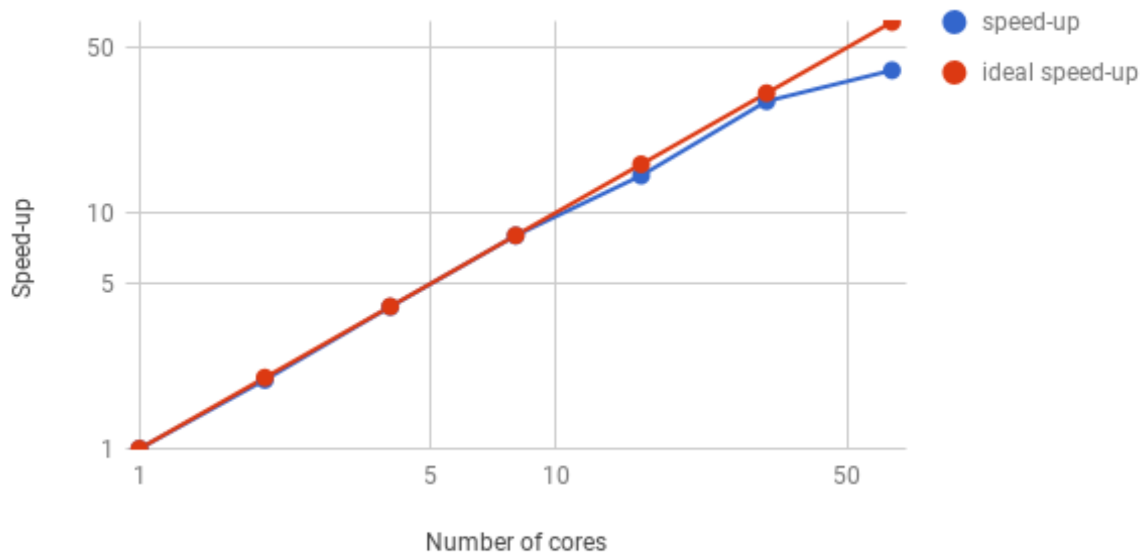


Figure 6.10: Speed-up of TRIP on a tall 131072×512 matrix with 2^{26} entries, up to 64 cores.

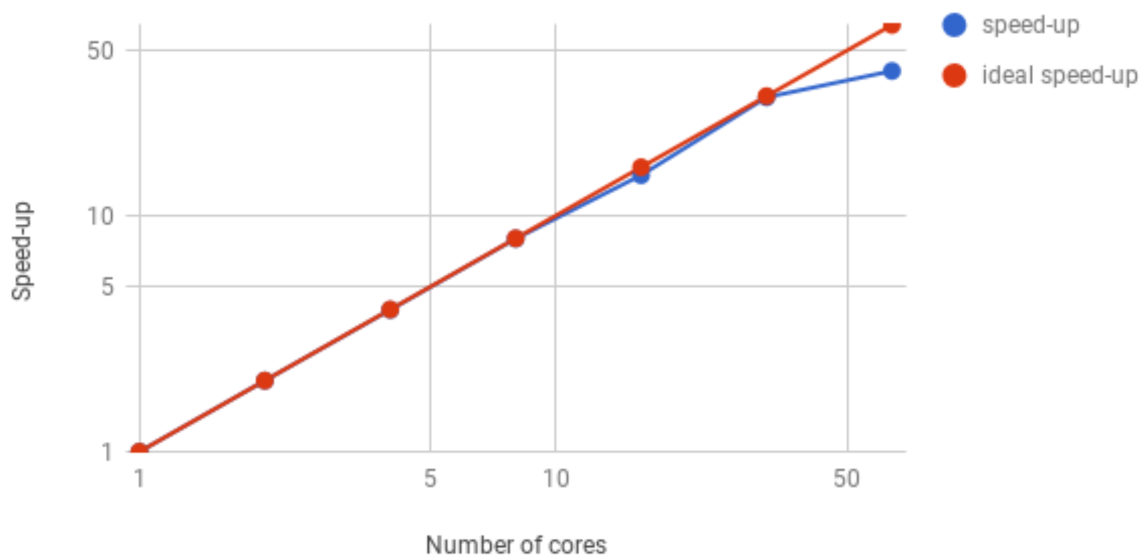


Figure 6.11: Speed-up of TRIP on a tall 33554432×2 matrix with 2^{26} entries, up to 64 cores.

7 Conclusions

TRIP is a highly parallel, in-place transpose algorithm for rectangular matrices that is—in contrast to classic algorithms—based on the divide-and-conquer principle. Its correctness has been proven, and computational complexity has been analyzed.

The correctness proof of the recursive definition of the algorithm gives freedom to the creators of concrete implementations. It is possible to apply the same arguments as outlined in the proof to modified versions of divide-and-conquer principle of TRIP; the correctness of optimized versions of TRIP can be derived from this proof.
One modification to the algorithm that decreases TRIP's work is sketched in Section 5.4.4.

Concrete implementations rely on iterators to allow the correct and efficient calculation of indices in sub-matrices. Examples of such iterators are presented in this thesis.

Furthermore the thesis offers an explanation of the "eagle curves", which are the graphical representations of work and span as functions of the matrix dimensions. These eagle curves cover the special case of matrices which dimensions are powers of two. In general the eagle curves can be interpreted as a lower bound of complexity for matrices with a given number of elements and varying aspect ratios. The shape of the eagle curves changes, when matrix dimensions that are no powers of two are taken into account. Depending on the divide strategy work and span can vary.

The complexity analysis in this thesis is based on bisection; a different strategy is hinted at in Section 5.4.4.

Future work may go into the topics of optimization for arbitrary matrix aspect ratios that are not powers of two, and general analysis for arbitrary aspect ratios.

References

[1] J. BENTLEY, *Programming Pearls*, ACM Press, 2013.

[2] J. BOOTHROYD, *Algorithm 302: Transpose Vector Stored Array*, Communications of the ACM, 10 (1967), pp. 292–293.

[3] N. BRENNER, *Algorithm 467: Matrix Transposition in Place [F1]*, Communications of the ACM, 16 (1973), pp. 692–694.

[4] E. G. CATE AND D. W. TWIGG, *Algorithm 513: Analysis of In-Situ Transposition [F1]*, Transactions on Mathematical Software (TOMS, 3 (1977), pp. 104–110.

[5] T. H. CORMEN, C. E. LEISERSON, R. L. RIVEST, AND C. STEIN, *Introduction to Algorithms*, MIT press, 3 ed., 2009.

[6] J. A. ELLIS AND M. MARKOV, *In Situ, Stable Merging by Way of the Perfect Shuffle*, The Computer Journal, 43 (2000), pp. 40–53.

[7] F. E. FICH, J. I. MUNRO, AND P. V. POBLETE, *Permuting in Place*, SIAM Journal on Computing, 24 (1995), pp. 266–278.

[8] M. FRIGO AND S. G. JOHNSON, *The Design and Implementation of FFTW3*, Proceedings of the IEEE, 93 (2005), pp. 216–231.

[9] F. G. GUSTAVSON AND T. SWIRSZCZ, *In-Place Transposition of Rectangular Matrices*, PARA, 4699 (2006), pp. 560–569.

[10] INTEL©, *Reference Manual for Intel© Math Kernel Library 11.3 - C*. https://software.intel.com/en-us/node/520862.

[11] P. JAIN, *A Simple In-Place Algorithm for In-Shuffle*. https://arxiv.org/pdf/0805.1598.pdf, May 2008.

[12] S. LAFLIN AND M. BREBNER, *Algorithm 380: In-Situ Transposition of a Rectangular Matrix [F1]*, Communications of the ACM, 13 (1970), pp. 324–326.

[13] OPENCFD LTD. (2009), *OpenFOAM – The Open Source CFD Toolbox*. www.openfoam.com.

[14] M. R. PORTNOFF, *An Efficient Parallel-Processing Method for Transposing Large Matrices in Place*, Image Processing, IEEE Transactions on, 8 (1999), pp. 1265–1275.

[15] C. E. L. ROBERT D. BLUMOFE, *Scheduling Multithreaded Computations by Work Stealing*, J. ACM, 46 (1994), pp. 720–748.

[16] T. B. SCHARDL, W. S. MOSES, AND C. E. LEISERSON, *Tapir: Embedding fork-join parallelism into llvm's intermediate representation*, in Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, ACM, 2017, pp. 249–265.

[17] I.-J. SUNG, J. GÓMEZ-LUNA, J. M. GONZÁLEZ-LINARES, N. GUIL, AND W.-M. W. HWU, *In-Place Transposition of Rectangular Matrices on Accelerators*, in ACM SIGPLAN Notices, vol. 49, ACM, 2014, pp. 207–218.

[18] D. VYUKOV, *Scalable Go Scheduler Design Doc*. https://docs.google.com/document/d/1TTj4T2JO42uD5ID9e89oa0sLKhJYD0Y_kqxDv3I3XMw, 5 2012.

[19] P. WINDLEY, *Transposing Matrices in a Digital Computer*, The Computer Journal, 2 (1959), pp. 47–48.

Appendix 1 - Code Listing - trip.h

```
#ifndef __TRIP
#define __TRIP

#include <stddef.h>
#include <stdint.h>

#define REVERSE_VOODOO 1
#define MIN(X, Y) (((X) < (Y)) ? (X) : (Y))
#define MAX(X, Y) (((X) < (Y)) ? (Y) : (X))

typedef struct {
  uint64_t work;
  uint64_t span;
} ws;

void next(size_t *i, size_t *count, size_t j0, size_t j1, size_t N, size_t p);
void prev(size_t *i, size_t *count, size_t j0, size_t j1, size_t N, size_t p);

ws reverse(float *a, size_t m0, size_t m1, size_t i0, size_t j0, size_t j1,
           size_t N);
ws reverse_recursive(float *a, size_t m0, size_t m1, size_t l, size_t i0,
                     size_t j0, size_t j1, size_t N);
ws reverse_offset(float *a, size_t m0, size_t m1, size_t l, size_t i0,
                  size_t j0, size_t j1, size_t N);
void swap_single(float *x, float *y);

ws merge(float *a, size_t p, size_t q, size_t i0, size_t i1, size_t j0,
         size_t j1, size_t N);
ws merger(float *a, size_t p, size_t q, size_t i0, size_t i1, size_t j0,
          size_t j1, size_t m0, size_t m1, size_t k, size_t N);
ws split(float *a, size_t p, size_t q, size_t i0, size_t i1, size_t j0,
         size_t j1, size_t N);
ws splitr(float *a, size_t p, size_t q, size_t i0, size_t i1, size_t j0,
```

```
        size_t j1, size_t s0, size_t s1, size_t k, size_t N);

ws transpose(float *a, size_t i0, size_t i1, size_t j0, size_t j1, size_t N);
ws transpose4(float *a, size_t I0, size_t J0, size_t i0, size_t i1, size_t j0,
          size_t j1, size_t N);

ws combineComplexity(ws a, ws b);
void addComplexity(ws *const a, ws b);
void combineAndAddComplexity(ws *const comp, ws a, ws b);
void combineAndAddComplexity4(ws *const comp, ws a, ws b, ws c, ws d);
void increaseComplexity(ws *const comp);

#endif
```

Appendix 2 - Code Listing - trip.c

```
#include "trip.h"
#include <cilk/cilk.h>

ws transpose(float *a, size_t i0, size_t i1, size_t j0, size_t j1, size_t N) {
  size_t m, n;
  ws comp = {1, 1};

  m = i1 - i0;
  n = j1 - j0;

  if ((m == 1) || (n == 1)) {
    return comp;
  }
  if (m == n) {
    ws t4Comp = transpose4(a, i0, j0, 0, m, 0, n, N);
    addComplexity(&comp, t4Comp);
    return comp;
  } else if (m > n) {
    size_t im;
    if (m < 2 * n)
      im = i0 + n;
    else
      im = (i1 + i0) / 2;

    ws tComp1 = cilk_spawn transpose(a, i0, im, j0, j1, N);
    ws tComp2 = cilk_spawn transpose(a, im, i1, j0, j1, N);
    cilk_sync;
    combineAndAddComplexity(&comp, tComp1, tComp2);

    ws mComp = merge(a, im - i0, i1 - im, i0, i1, j0, j1, N);
    addComplexity(&comp, mComp);
    return comp;
  } else {  // (m < n)
```

```
    size_t jm;
    if (2 * m > n)
      jm = j0 + m;
    else
      jm = (j1 + j0) / 2;

    ws tComp1 = cilk_spawn transpose(a, i0, i1, j0, jm, N);
    ws tComp2 = cilk_spawn transpose(a, i0, i1, jm, j1, N);
    cilk_sync;
    combineAndAddComplexity(&comp, tComp1, tComp2);

    ws sComp = split(a, jm - j0, j1 - jm, i0, i1, j0, j1, N);
    addComplexity(&comp, sComp);
    return comp;
  }
  return comp;
}

ws split(float *a, size_t p, size_t q, size_t i0, size_t i1, size_t j0,
         size_t j1, size_t N) {
  size_t s0, s1;

  s0 = 0;
  s1 = (j1 - j0) * (i1 - i0);

  return splitr(a, p, q, i0, i1, j0, j1, s0, s1, i1 - i0, N);
}

ws splitr(float *a, size_t p, size_t q, size_t i0, size_t i1, size_t j0,
          size_t j1, size_t s0, size_t s1, size_t k, size_t N) {
  size_t k2 = k / 2;
  size_t rm, r0, r1;
  size_t sm;
  ws comp = {1, 1};

  if (k == 1) return comp;

  // split left and right part
  sm = s0 + k2 * (p + q);
  ws sComp1 = cilk_spawn splitr(a, p, q, i0, i1, j0, j1, s0, sm, k2, N);
```

```
  ws sComp2 = cilk_spawn splitr(a, p, q, i0, i1, j0, j1, sm, s1, k - k2, N);
  cilk_sync;
  combineAndAddComplexity(&comp, sComp1, sComp2);

  // rotate middle part
  r0 = s0 + k2 * p;
  r1 = s0 + k * p + k2 * q;

  ws rComp = cilk_spawn reverse(a, r0, r1, i0, j0, j1, N);
  cilk_sync;
  addComplexity(&comp, rComp);

  // rotate left and right part
  rm = s0 + k * p;
  ws rComp1 = cilk_spawn reverse(a, r0, rm, i0, j0, j1, N);
  ws rComp2 = cilk_spawn reverse(a, rm, r1, i0, j0, j1, N);
  cilk_sync;
  combineAndAddComplexity(&comp, rComp1, rComp2);

  return comp;
}

void next(size_t *i, size_t *count, size_t j0, size_t j1, size_t N, size_t p) {
  if (*count == p - 1) {
    *count = 0;
    *i += (N - j1) + j0 + 1;
  } else {
    *count += 1;
    *i += 1;
  }
}

void prev(size_t *i, size_t *count, size_t j0, size_t j1, size_t N, size_t p) {
  if (*count == 0) {
    *count = p - 1;
    *i -= j0 + (N - j1) + 1;
  } else {
    *count -= 1;
    *i -= 1;
  }
```

```
}

/
   a .... array
   i0 ... vertical offset of sub−matrix
   j0 ... horizontal offset of sub−matrix
   j1 ... horizontal offset of end of sub−matrix; j0 to j1 is the span of the
   matrix
   N .... width of original matrix
   m0 ... left boundary index of array to be reversed
   m1 ... right boundary index of array to be reversed
   l .... parallelization index .. reverse_offset will swap between
   m0 and l, and m1−l and m1
  /
ws reverse_offset(float *a, size_t m0, size_t m1, size_t l, size_t i0,
                  size_t j0, size_t j1, size_t N) {
  size_t i, next_count;
  size_t j, prev_count;
  size_t m, mm;
  size_t p;
  float tmp;
  ws comp = {0, 0};

  p = j1 - j0;

  // index starting from left (going right); original matrix index
  i = i0 * N + j0 + (m0 / p) * N + (m0 % p);
  next_count = m0 % p;

  // index starting from right (going left); original matrix index
  j = i0 * N + j0 + ((m1 - 1) / p) * N + ((m1 - 1) % p);
  prev_count = (m1 - 1) % p;

  mm = m0 + l;
  for (m = m0; m < mm; next(&i, &next_count, j0, j1, N, p),
       prev(&j, &prev_count, j0, j1, N, p), m++) {
    tmp = a[j];
    a[j] = a[i];
    a[i] = tmp;
    increaseComplexity(&comp);
```

```
  }
  return comp;
}

ws reverse(float *a, size_t m0, size_t m1, size_t i0, size_t j0, size_t j1,
           size_t N) {
  return reverse_recursive(a, m0, m1, (m1 - m0) / 2, i0, j0, j1, N);
}

/
   reverses a sub-list in a block in a matrix.
   a, i0, j0, j1, p, N ... non-parallelizing parameters
   a .... array
   m0 ... starting index of sub-array to be reversed
   m1 ... end-index of sub-array to be reversed (exclusive)
   l .... parallelization parameter (use (m1-m0)/2 to initialize)
          length of sub-array to be swapped with right part of array
   i0 ... offset of sub-matrix in vertical direction
   j0 ... offset of sub-matrix in horizontal direction
   j1 ... offset of end of sub-matrix in horizontal direction (exclusive)
   N .... width of original matrix
 /
ws reverse_recursive(float *a, size_t m0, size_t m1, size_t l, size_t i0,
                     size_t j0, size_t j1, size_t N) {
  ws comp = {1, 1};

  if (l > REVERSE_VOODOO) {
    size_t lm = l / 2;

    ws rComp1 = cilk_spawn reverse_recursive(a, m0, m1, lm, i0, j0, j1, N);
    ws rComp2 = cilk_spawn reverse_recursive(a, m0 + lm, m1 - lm, l - lm, i0,
                                             j0, j1, N);
    cilk_sync;
    combineAndAddComplexity(&comp, rComp1, rComp2);
  } else {
    ws rComp = reverse_offset(a, m0, m1, l, i0, j0, j1, N);
    addComplexity(&comp, rComp);
  }
  return comp;
}
```

```
/
   merges the block defined by the boundaries [i0, i1[ and [j0, j1[
  /
ws merge(float *a, size_t p, size_t q, size_t i0, size_t i1, size_t j0,
         size_t j1, size_t N) {
  size_t m0, m1;

  m0 = 0;
  m1 = (j1 - j0) * (i1 - i0);
  return merger(a, p, q, i0, i1, j0, j1, m0, m1, j1 - j0, N);
}

/
   a .... array
   p .... number of upper rows to be merged
   q .... number of lower rows to be merged
   i0 ... starting row of sub-matrix (inclusive)
   i1 ... ending row of sub-matrix (exclusive)
   j0 ... starting column of sub-matrix (inclusive)
   j1 ... ending column of sub-matrix (exclusive)
   m0 ... starting index of the sub-part of the array to be merged (inclusive)
   m1 ... ending index of the sub-part of the array to be merged (exclusive)
   k .... termination parameter
   N .... number of columns in the original big matrix (for iterators in
   'reverse')
  /
ws merger(float *a, size_t p, size_t q, size_t i0, size_t i1, size_t j0,
          size_t j1, size_t m0, size_t m1, size_t k, size_t N) {
  size_t k2 = k / 2;  // == k/2
  size_t rm, r0, r1;  // reversal middle- and end-posize_ts
  size_t mm;
  ws comp = {1, 1};

  if (k == 1) return comp;

  // for rotation first reverse middle part
  // then reverse left and right

  // first reverse whole middle part
```

```
  r0 = m0 + k2 * p;
  r1 = m0 + k * p + k2 * q;

  ws rComp = reverse(a, r0, r1, i0, j0, j1, N);
  addComplexity(&comp, rComp);

  // then reverse left and right of the middle part
  rm = r0 + k2 * q;
  ws rComp1 = cilk_spawn reverse(a, r0, rm, i0, j0, j1, N);
  ws rComp2 = cilk_spawn reverse(a, rm, r1, i0, j0, j1, N);
  cilk_sync;
  combineAndAddComplexity(&comp, rComp1, rComp2);

  // now merge the resulting sub-arrays
  mm = m0 + k2 * (p + q);   // == rm
  ws mComp1 = cilk_spawn merger(a, p, q, i0, i1, j0, j1, m0, mm, k2, N);
  ws mComp2 =
      cilk_spawn merger(a, p, q, i0, i1, j0, j1, mm, m1, k - k2,
                        N);  // k-k2 so not both k2's are 1 in case of e.g. k==3
  cilk_sync;
  combineAndAddComplexity(&comp, mComp1, mComp2);

  return comp;
}

/
   a .... array
   I0 ... upper starting row of sub-matrix (inclusive)
   J0 ... left starting column of sub-matrix (inclusive)
   i0 ... upper starting row of parallel partition of sub-matrix to be
   transposed (inclusive)
   i1 ... lower ending row of parallel partition of sub-matrix to be transposed
   (exclusive)
   j0 ... left starting column of parallel partition of sub-matrix to be
   transposed (inclusive)
   j1 ... right ending column of parallel partition of sub-matrix to be
   transposed (exclusive)
   N .... number of columns of original matrix
  /
ws transpose4(float *a, size_t I0, size_t J0, size_t i0, size_t i1, size_t j0,
```

```
              size_t j1, size_t N) {
  ws comp = {1, 1};

  if (i1 - i0 > 1) {
    size_t im = (i0 + i1) / 2, jm = (j0 + j1) / 2;
    ws tComp1, tComp2, tComp3, tComp4 = {0, 0};
    tComp1 = cilk_spawn transpose4(a, I0, J0, i0, im, j0, jm, N);
    tComp2 = cilk_spawn transpose4(a, I0, J0, i0, im, jm, j1, N);
    tComp3 = cilk_spawn transpose4(a, I0, J0, im, i1, jm, j1, N);
    if (i1 <= j0) tComp4 = cilk_spawn transpose4(a, I0, J0, im, i1, j0, jm, N);
    cilk_sync;
    combineAndAddComplexity4(&comp, tComp1, tComp2, tComp3, tComp4);

  } else {
    size_t j;
    for (j = j0; j < j1; j++) {
      swap_single(&a[(I0 + j) * N + J0 + i0], &a[(I0 + i0) * N + J0 + j]);
      increaseComplexity(&comp);
    }
  }
  return comp;
}

void swap_single(float *x, float *y) {
  float t = *x;
  *x = *y;
  *y = t;
}

ws combineComplexity(ws a, ws b) {
  ws result;

  result.span = MAX(a.span, b.span);
  result.work = a.work + b.work;

  return result;
}

void addComplexity(ws *const comp, ws b) {
  comp->span += b.span;
```

```
  comp->work += b.work;
}

void combineAndAddComplexity(ws *const comp, ws a, ws b) {
  ws combined = combineComplexity(a, b);
  addComplexity(comp, combined);
}

void combineAndAddComplexity4(ws *const comp, ws a, ws b, ws c, ws d) {
  ws ab = combineComplexity(a, b);
  ws cd = combineComplexity(c, d);
  ws abcd = combineComplexity(ab, cd);
  addComplexity(comp, abcd);
}

void increaseComplexity(ws *const comp) {
  comp->span += 1;
  comp->work += 1;
}
```