Technische Universität Braunschweig

Algorithmen und Datenstrukturen – Übung #9

Exkurs: 2-dimensionales Paralleles Sortieren

Matthias Konitzny, Arne Schmidt

10.02.22

Sequentiell vs Parallel

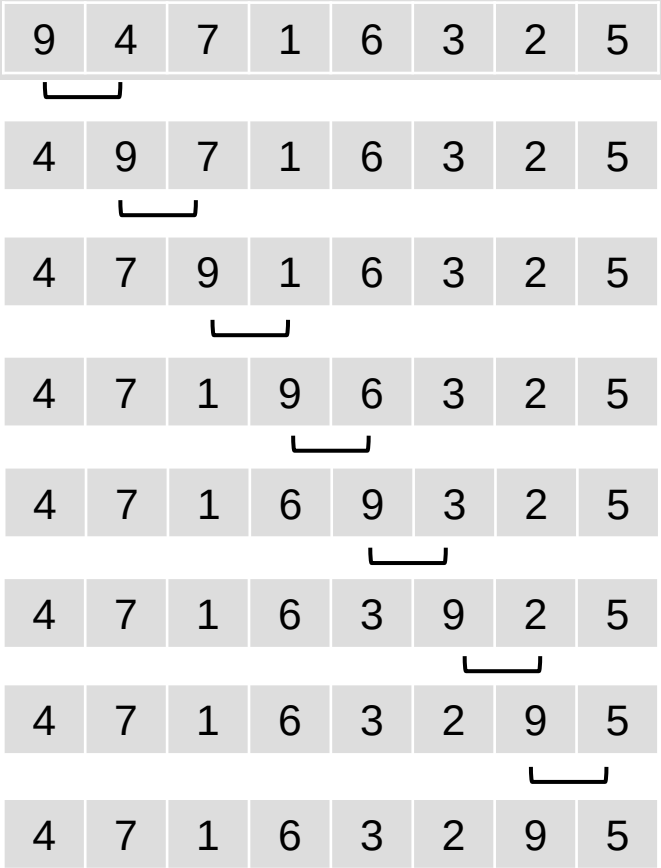
```
1: function BubbleSort(A)
2:     for i = n downto 2 do
3:         for j = 1 to i − 1 do
4:             if A[j] > A[j + 1] then
5:                 Vertausche A[j]und A[j + 1]
```

Algorithmus 1: Der BubbleSort-Algorithmus.

In jeder Iteration wird die größte Zahl im verbliebenen Teilarray an die letzte Position gebracht.

Laufzeit: $O(n^2)$

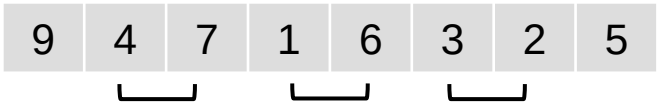
Idee: Parallelisiere Vergleiche



Sequentiell vs Parallel

Idee: Parallelisiere Vergleiche

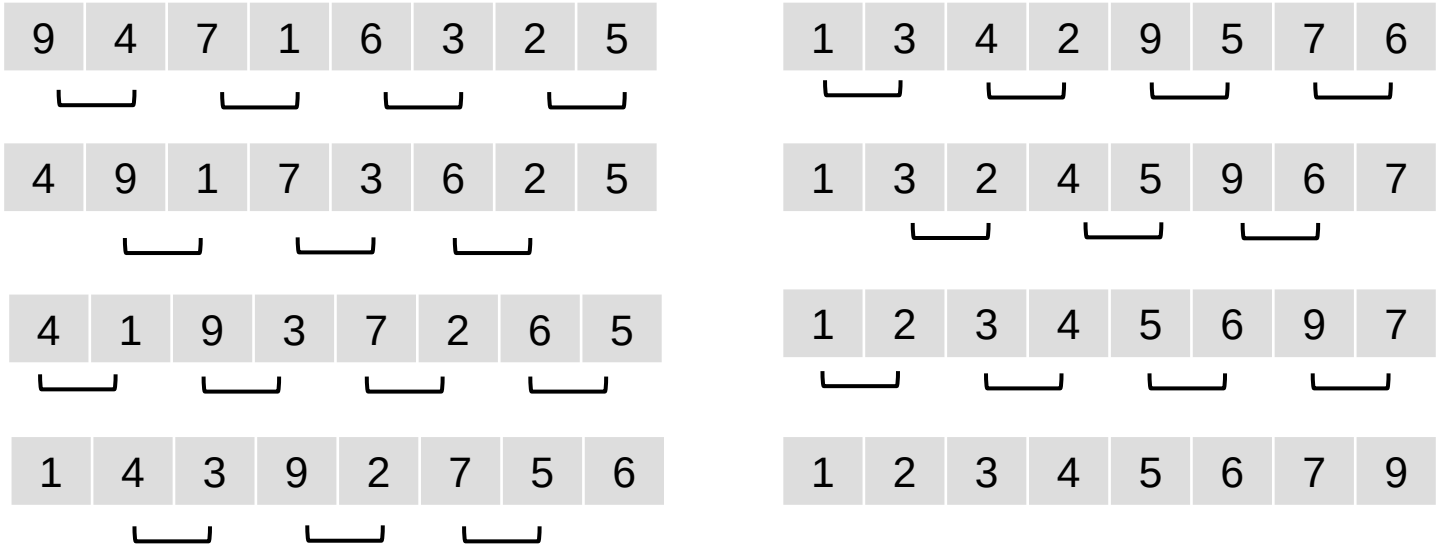
Gerade Positionen



Ungerade Positionen

Parallelles Bubblesort (Odd-Even Transposition Sort)

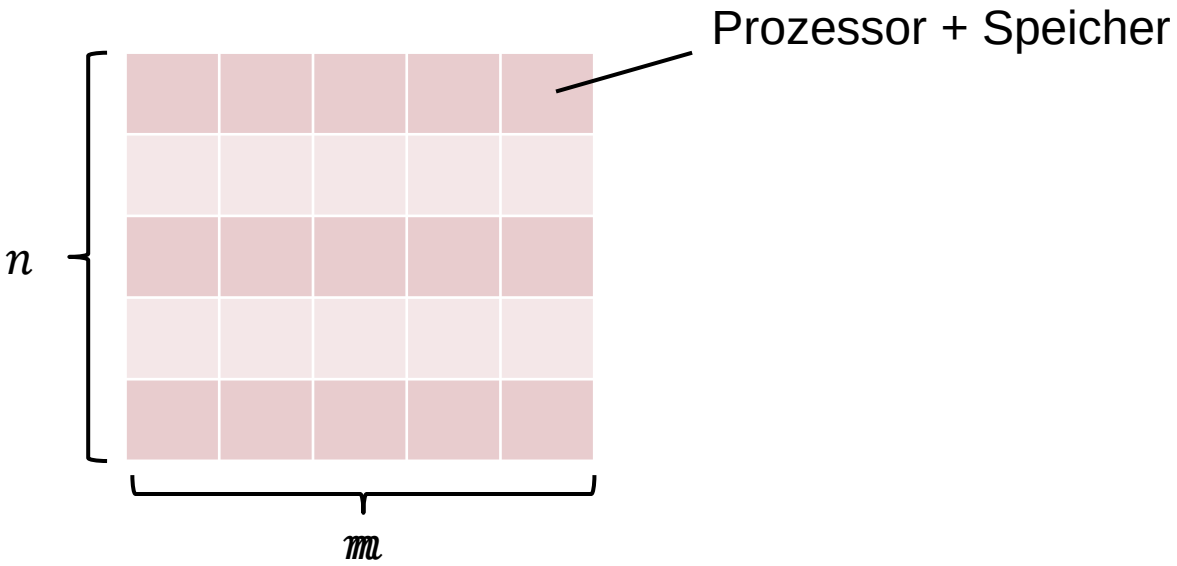
Idee: Sortiere abwechselnd nach geraden und ungeraden Positionen



Laufzeit: $O(n)$

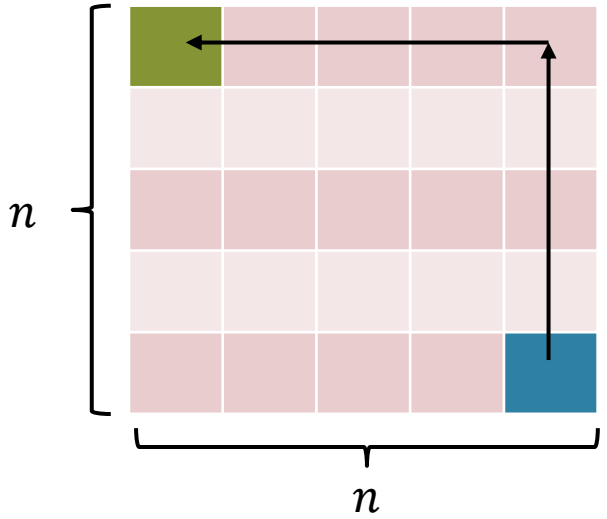
Parallele Berechnungen – In 2D

Neues Rechenmodell: Mesh-Connected Computer



Anwendung: VLSI Chipdesign für sehr schnelle Hardwarebeschleunigung

Paralleles Sortieren – Untere Schranke



Untere Schranke: $2n - 2$

[Kun 87]: $3n - \sqrt{2n} - 3$

Kunde, M. Lower bounds for sorting on mesh-connected architectures. *Acta Informatica* **24,** 121–130 (1987). https://doi.org/10.1007/BF00264359

Paralleles Sortieren

9	6	4
1	8	3
5	2	7

Idee: Sortiere abwechselnd Zeilen und Spalten

4	6	9
1	3	8
2	5	7

Zeilen

1	3	7
2	5	8
4	6	9

Spalten

1	3	7
2	5	8
4	6	9

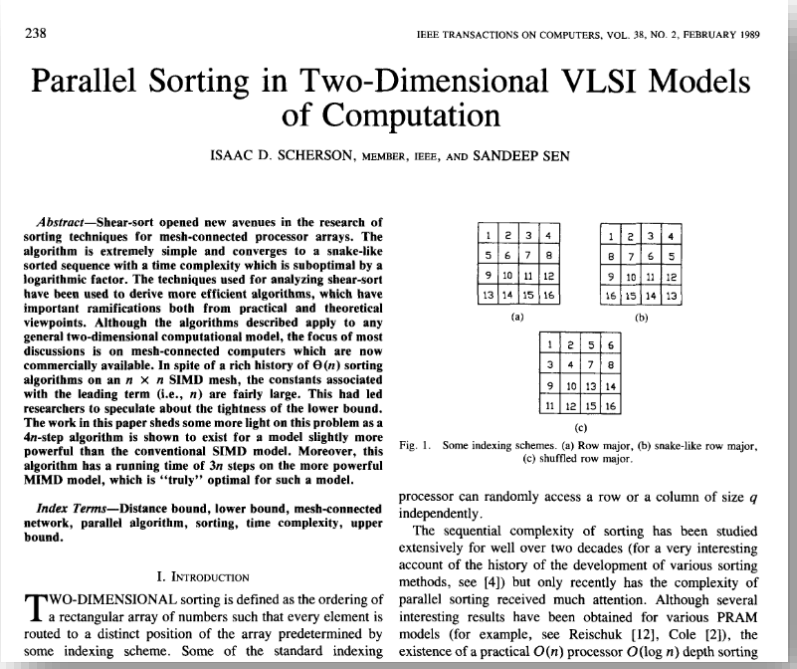
Zeilen

Problem: Größere Zahlen kommen nicht nach unten und kleinere nicht nach oben.

Shearsort

Algorithmus SHEARSORT

1. **for** i **in** $1 \ldots \log(n)$ **do**
2. Sortiere die Zeilen (Alternierend)
3. Sortiere die Spalten
4. Sortiere die Zeilen



238 IEEE TRANSACTIONS ON COMPUTERS, VOL. 38, NO. 2, FEBRUARY 1989

Parallel Sorting in Two-Dimensional VLSI Models of Computation

ISAAC D. SCHERSON, MEMBER, IEEE, AND SANDEEP SEN

Abstract—**Shear-sort opened new avenues in the research of sorting techniques for mesh-connected processor arrays. The algorithm is extremely simple and converges to a snake-like sorted sequence with a time complexity which is suboptimal by a logarithmic factor. The techniques used for analyzing shear-sort have been used to derive more efficient algorithms, which have important ramifications both from practical and theoretical viewpoints. Although the algorithms described apply to any general two-dimensional computational model, the focus of most discussions is on mesh-connected computers which are now commercially available. In spite of a rich history of $\Theta(n)$ sorting algorithms on an $n \times n$ SIMD mesh, the constants associated with the leading term (i.e., n) are fairly large. This had led researchers to speculate about the tightness of the lower bound. The work in this paper sheds some more light on this problem as a $4n$-step algorithm is shown to exist for a model slightly more powerful than the conventional SIMD model. Moreover, this algorithm has a running time of $3n$ steps on the more powerful MIMD model, which is "truly" optimal for such a model.**

Index Terms—**Distance bound, lower bound, mesh-connected network, parallel algorithm, sorting, time complexity, upper bound.**

I. INTRODUCTION

TWO-DIMENSIONAL sorting is defined as the ordering of a rectangular array of numbers such that every element is routed to a distinct position of the array predetermined by some indexing scheme. Some of the standard indexing

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

(a)

1	2	3	4
8	7	6	5
9	10	11	12
16	15	14	13

(b)

1	2	5	6
3	4	7	8
9	10	13	14
11	12	15	16

(c)

Fig. 1. Some indexing schemes. (a) Row major, (b) snake-like row major, (c) shuffled row major.

processor can randomly access a row or a column of size q independently.

The sequential complexity of sorting has been studied extensively for well over two decades (for a very interesting account of the history of the development of various sorting methods, see [4]) but only recently has the complexity of parallel sorting received much attention. Although several interesting results have been obtained for various PRAM models (for example, see Reischuk [12], Cole [2]), the existence of a practical $O(n)$ processor $O(\log n)$ depth sorting

Scherson, Isaac D., and Sandeep Sen. "Parallel sorting in two-dimensional VLSI models of computation." *IEEE Transactions on Computers* 38.2 (1989): 238-249.

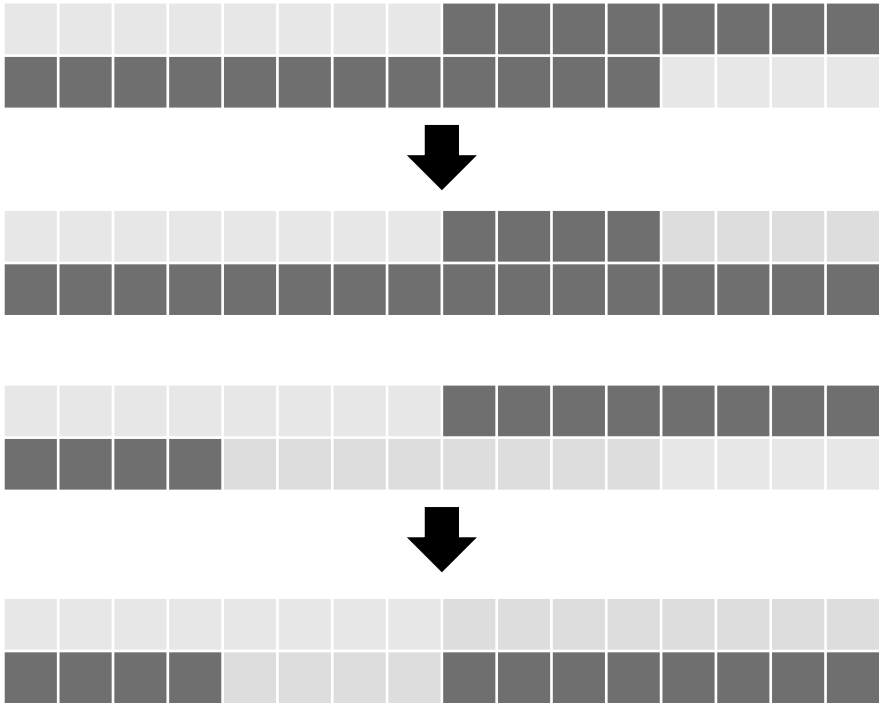
Shearsort

Algorithmus SHEARSORT

1. **for** i **in** $1 \ldots \log(n)$ **do**
2. Sortiere die Zeilen (Alternierend)
3. Sortiere die Spalten
4. Sortiere die Zeilen

Jeder Durchlauf halbiert die Anzahl der unsortierten Zeilen

Laufzeit: $O(n \log n)$



Scherson, Isaac D., and Sandeep Sen. "Parallel sorting in two-dimensional VLSI models of computation." *IEEE Transactions on Computers* 38.2 (1989): 238-249.

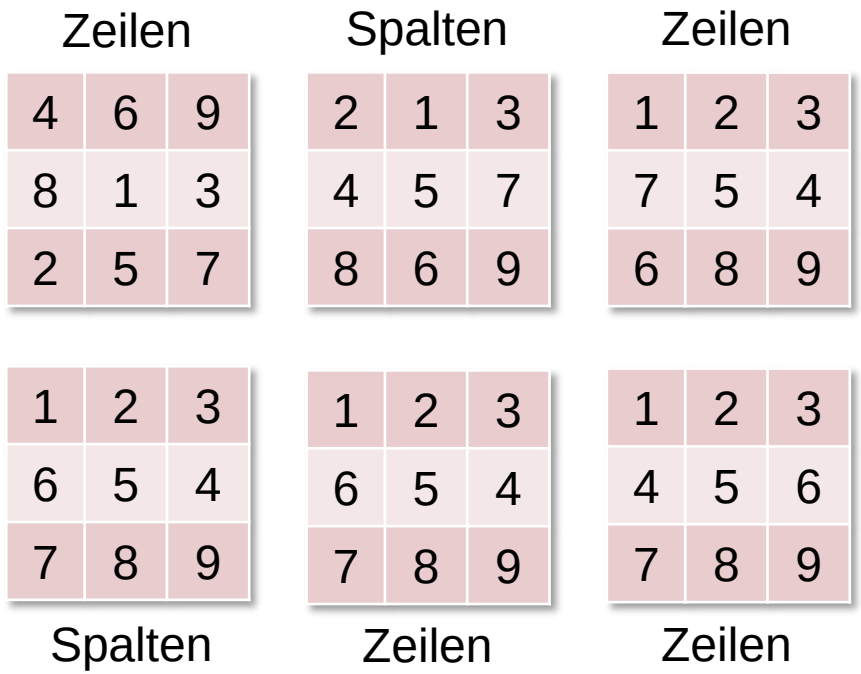
Shearsort

Algorithmus SHEARSORT

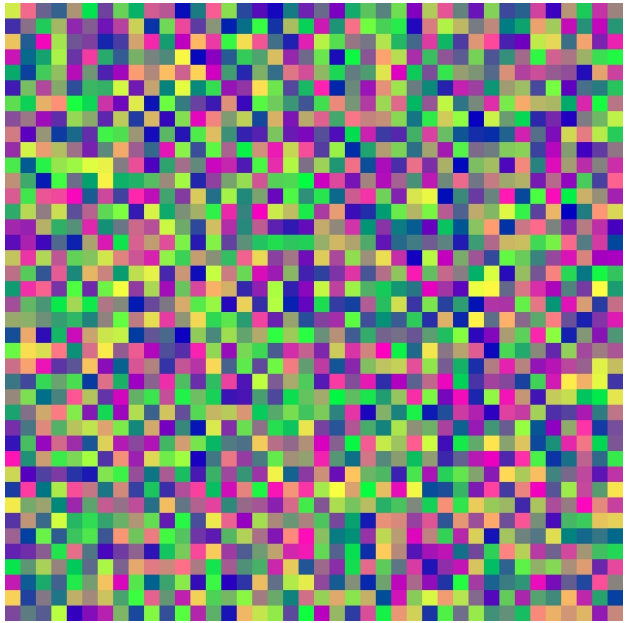
1. **for** i **in** $1 \dots \log(n)$ **do**
2. Sortiere die Zeilen (Alternierend)
3. Sortiere die Spalten
4. Sortiere die Zeilen

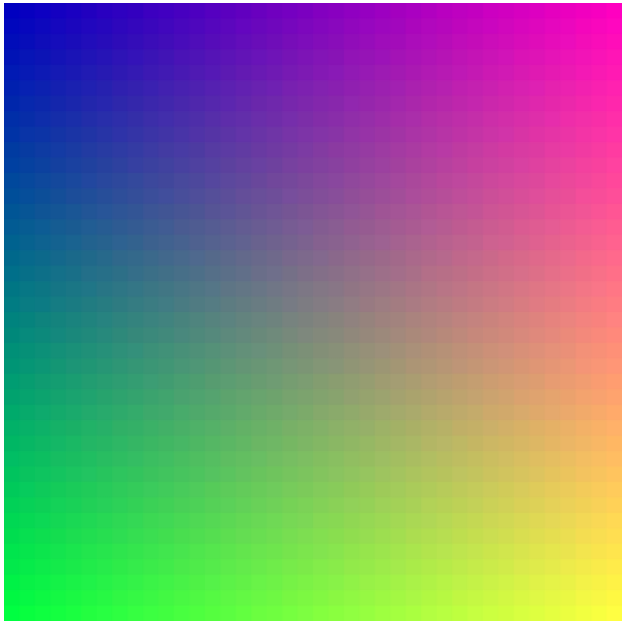
Jeder Durchlauf halbiert die Anzahl der unsortierten Zeilen

Laufzeit: $O(n \log n)$

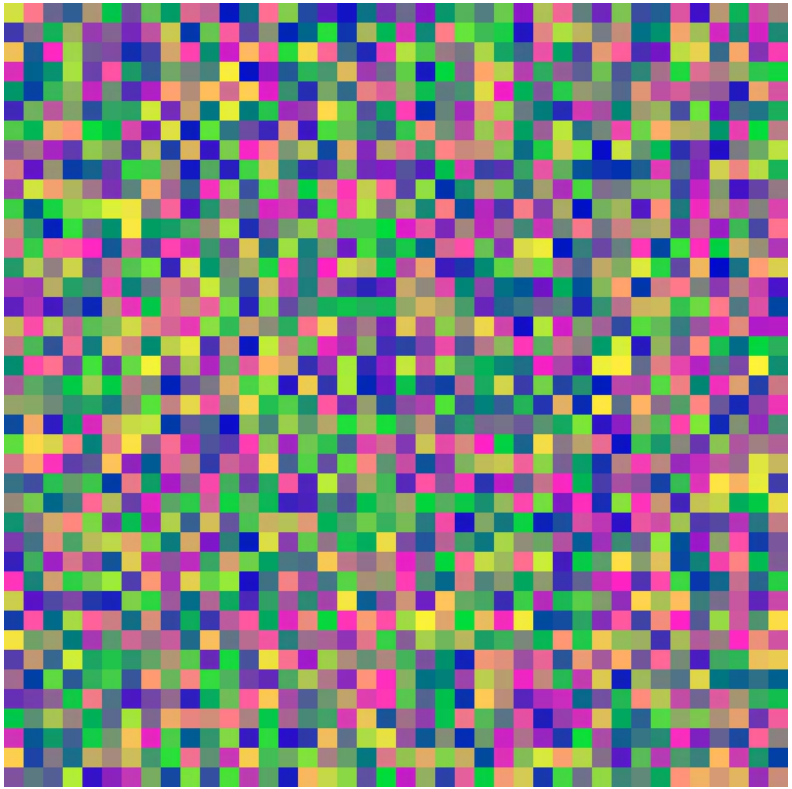


Shearsort in Aktion





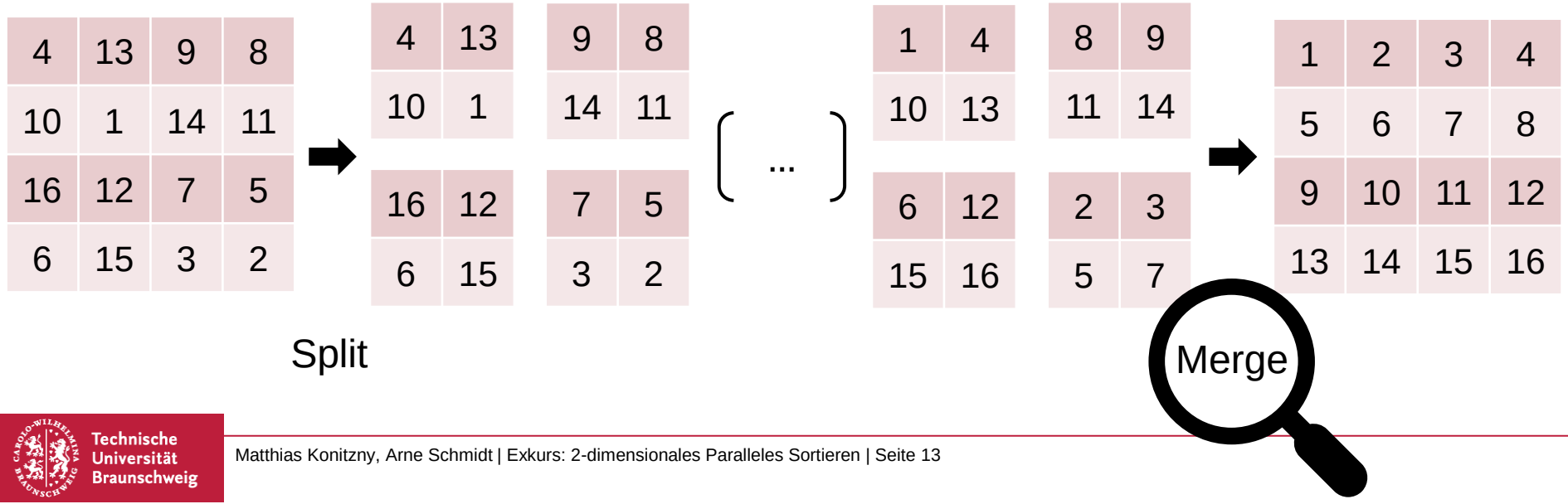
Shearsort in Aktion



4-Way Mergesort

Ziel: 2-dimensionales paralleles Sortieren in linearer Zeit

Idee: Nutze bekanntes Prinzip aus Mergesort



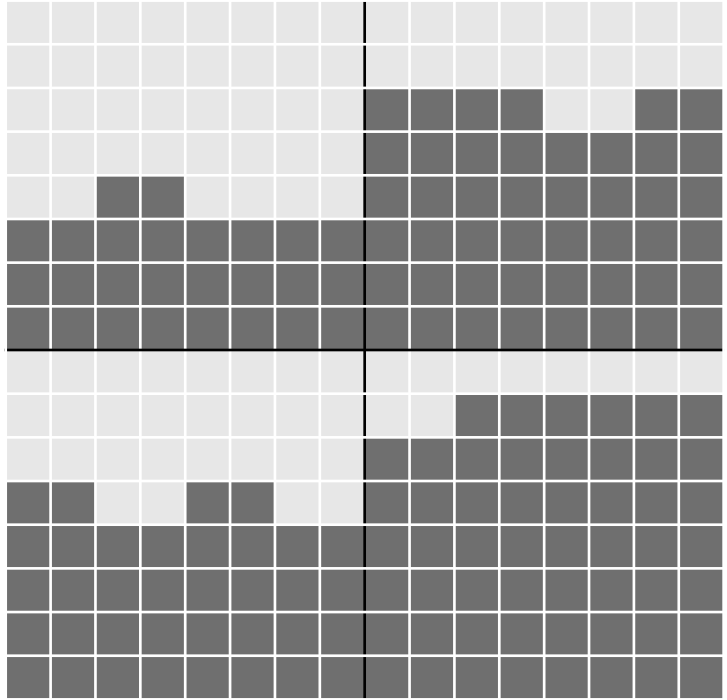
4-Way Mergesort

Algorithmus 4-WAY-MERGE

Eingabe: $k \times k$ Array, dessen $\frac{k}{2} \times \frac{k}{2}$ Teilarrays teilsortiert sind.

Ausgabe: Teilsortiertes $k \times k$ Array

1. Sortiere die Zeilen der *oberen* Teilarrays aufsteigend
2. Sortiere die Zeilen der *unteren* Teilarrays absteigend
3. Sortiere die Spalten
4. Sortiere die Zeilen der *ungeraden* Zeilen aufsteigend
5. Sortiere die Zeilen der *geraden* Zeilen absteigend
6. Sortiere die Spalten



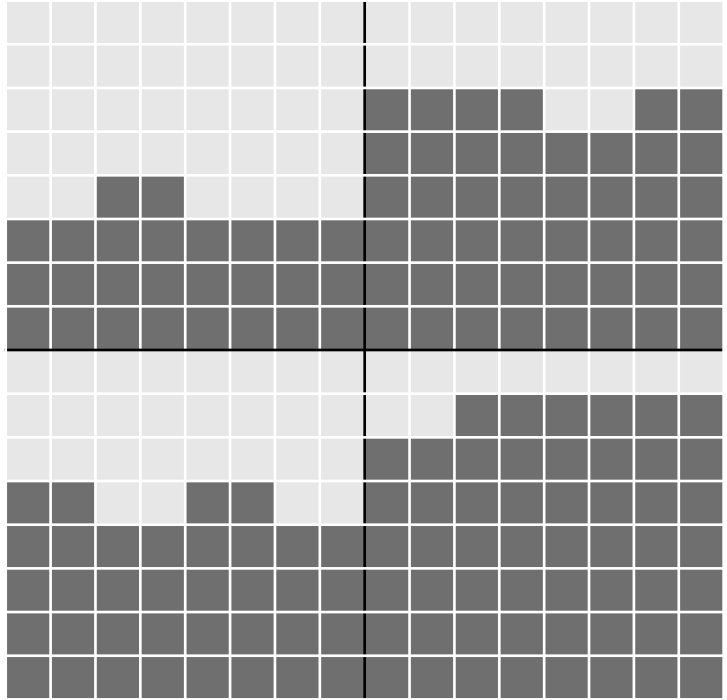
4-Way Mergesort

Algorithmus 4-WAY-MERGE

Eingabe: $k \times k$ Array, dessen $\frac{k}{2} \times \frac{k}{2}$ Teilarrays teilsortiert sind.

Ausgabe: Teilsortiertes $k \times k$ Array

1. Sortiere die Zeilen der *oberen* Teilarrays aufsteigend
2. Sortiere die Zeilen der *unteren* Teilarrays absteigend
3. Sortiere die Spalten
4. Sortiere die Zeilen der *ungeraden* Zeilen aufsteigend
5. Sortiere die Zeilen der *geraden* Zeilen absteigend
6. Sortiere die Spalten



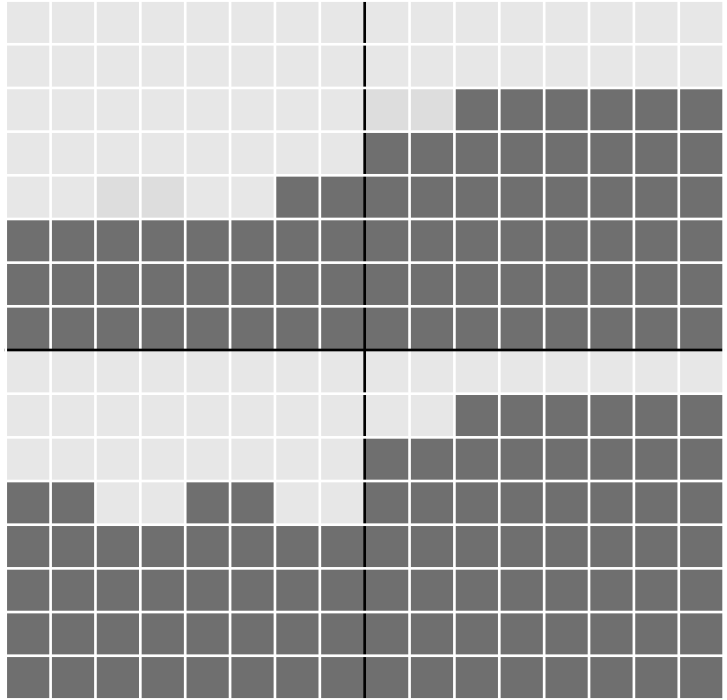
4-Way Mergesort

Algorithmus 4-WAY-MERGE

Eingabe: $k \times k$ Array, dessen $\frac{k}{2} \times \frac{k}{2}$ Teilarrays teilsortiert sind.

Ausgabe: Teilsortiertes $k \times k$ Array

1. Sortiere die Zeilen der *oberen* Teilarrays aufsteigend
2. Sortiere die Zeilen der *unteren* Teilarrays absteigend
3. Sortiere die Spalten
4. Sortiere die Zeilen der *ungeraden* Zeilen aufsteigend
5. Sortiere die Zeilen der *geraden* Zeilen absteigend
6. Sortiere die Spalten



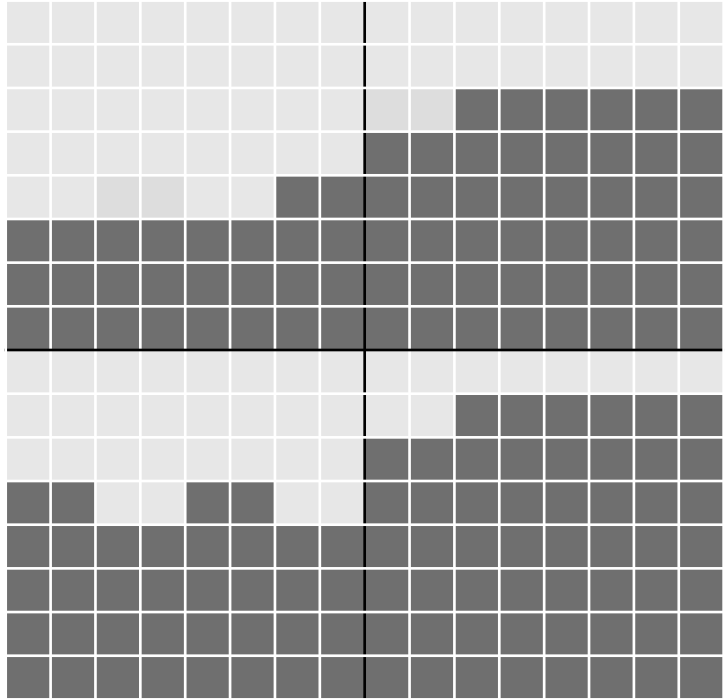
4-Way Mergesort

Algorithmus 4-WAY-MERGE

Eingabe: $k \times k$ Array, dessen $\frac{k}{2} \times \frac{k}{2}$ Teilarrays teilsortiert sind.

Ausgabe: Teilsortiertes $k \times k$ Array

1. Sortiere die Zeilen der *oberen* Teilarrays aufsteigend
2. Sortiere die Zeilen der *unteren* Teilarrays absteigend
3. Sortiere die Spalten
4. Sortiere die Zeilen der *ungeraden* Zeilen aufsteigend
5. Sortiere die Zeilen der *geraden* Zeilen absteigend
6. Sortiere die Spalten



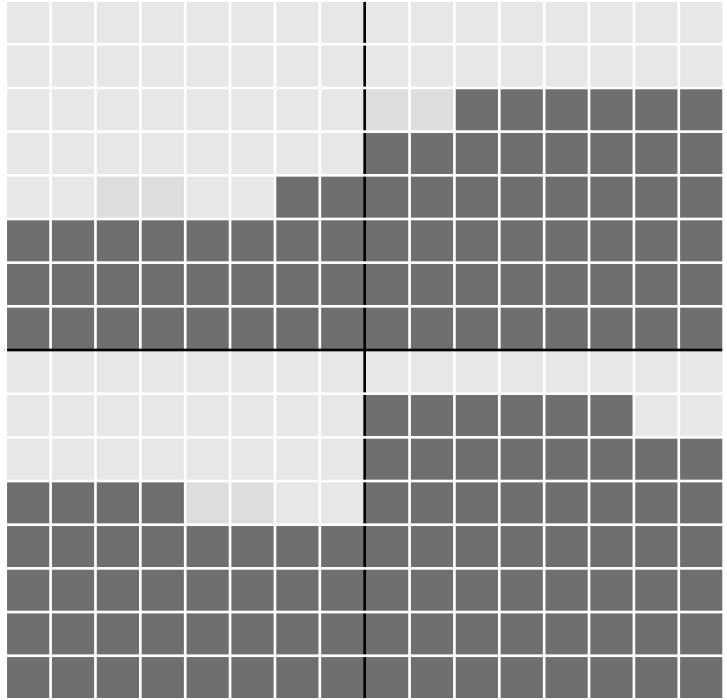
4-Way Mergesort

Algorithmus 4-WAY-MERGE

Eingabe: $k \times k$ Array, dessen $\frac{k}{2} \times \frac{k}{2}$ Teilarrays teilsortiert sind.

Ausgabe: Teilsortiertes $k \times k$ Array

1. Sortiere die Zeilen der *oberen* Teilarrays aufsteigend
2. Sortiere die Zeilen der *unteren* Teilarrays absteigend
3. Sortiere die Spalten
4. Sortiere die Zeilen der *ungeraden* Zeilen aufsteigend
5. Sortiere die Zeilen der *geraden* Zeilen absteigend
6. Sortiere die Spalten



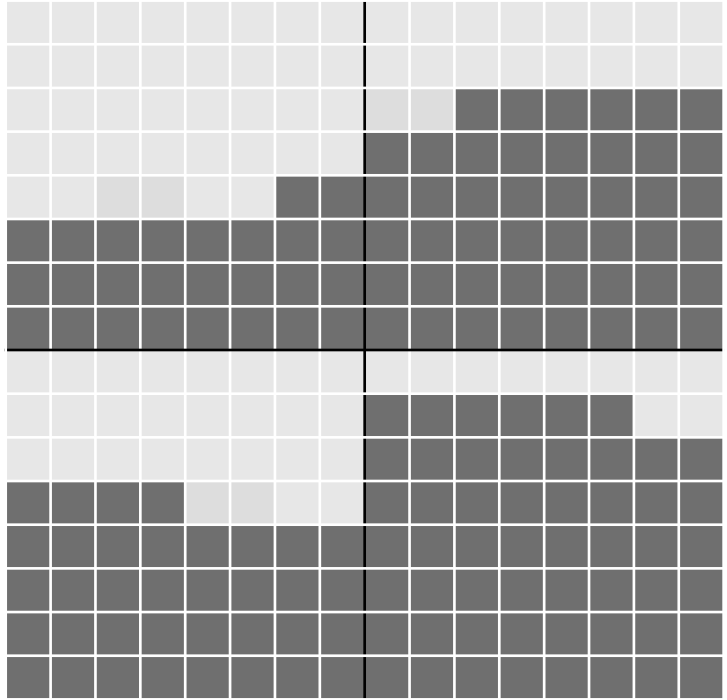
4-Way Mergesort

Algorithmus 4-WAY-MERGE

Eingabe: $k \times k$ Array, dessen $\frac{k}{2} \times \frac{k}{2}$ Teilarrays teilsortiert sind.

Ausgabe: Teilsortiertes $k \times k$ Array

1. Sortiere die Zeilen der *oberen* Teilarrays aufsteigend
2. Sortiere die Zeilen der *unteren* Teilarrays absteigend
3. Sortiere die Spalten
4. Sortiere die Zeilen der *ungeraden* Zeilen aufsteigend
5. Sortiere die Zeilen der *geraden* Zeilen absteigend
6. Sortiere die Spalten



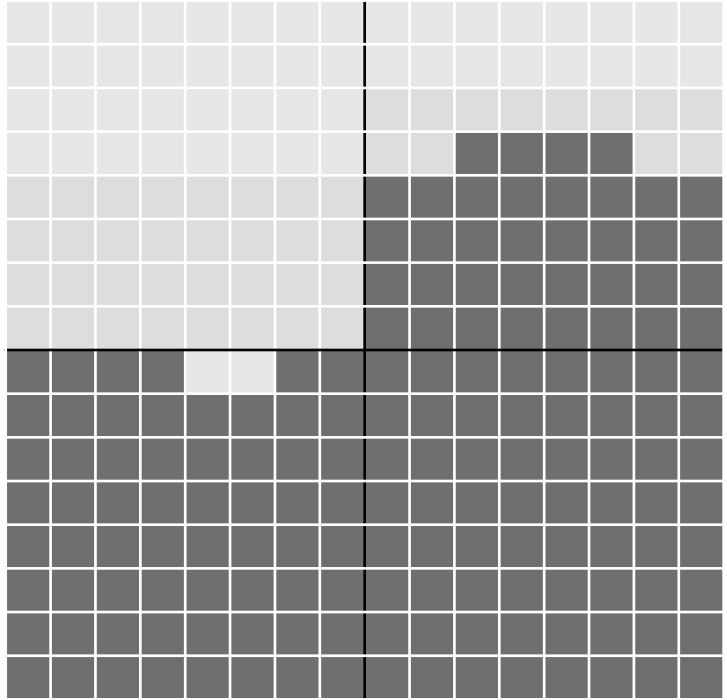
4-Way Mergesort

Algorithmus 4-WAY-MERGE

Eingabe: $k \times k$ Array, dessen $\frac{k}{2} \times \frac{k}{2}$ Teilarrays teilsortiert sind.

Ausgabe: Teilsortiertes $k \times k$ Array

1. Sortiere die Zeilen der *oberen* Teilarrays aufsteigend
2. Sortiere die Zeilen der *unteren* Teilarrays absteigend
3. Sortiere die Spalten
4. Sortiere die Zeilen der *ungeraden* Zeilen aufsteigend
5. Sortiere die Zeilen der *geraden* Zeilen absteigend
6. Sortiere die Spalten



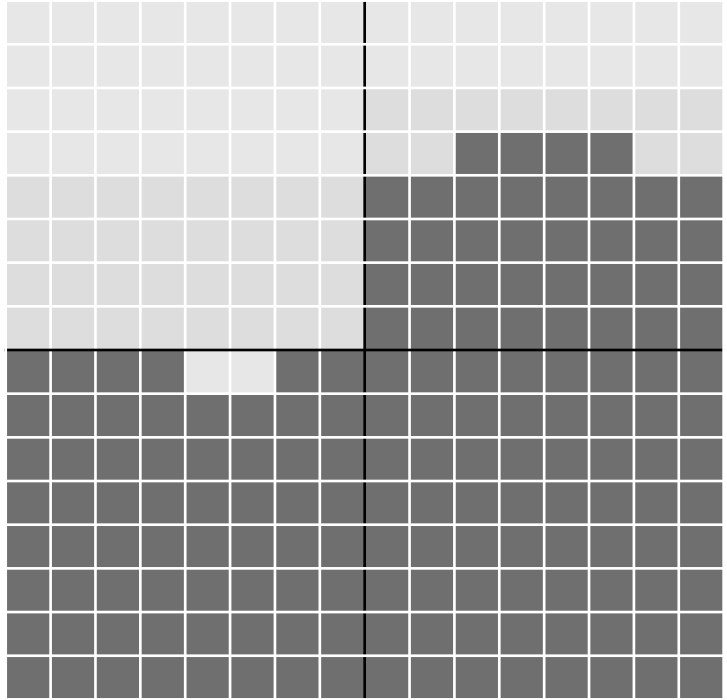
4-Way Mergesort

Algorithmus 4-WAY-MERGE

Eingabe: $k \times k$ Array, dessen $\frac{k}{2} \times \frac{k}{2}$ Teilarrays teilsortiert sind.

Ausgabe: Teilsortiertes $k \times k$ Array

1. Sortiere die Zeilen der *oberen* Teilarrays aufsteigend
2. Sortiere die Zeilen der *unteren* Teilarrays absteigend
3. Sortiere die Spalten
4. Sortiere die Zeilen der *ungeraden* Zeilen aufsteigend
5. Sortiere die Zeilen der *geraden* Zeilen absteigend
6. Sortiere die Spalten



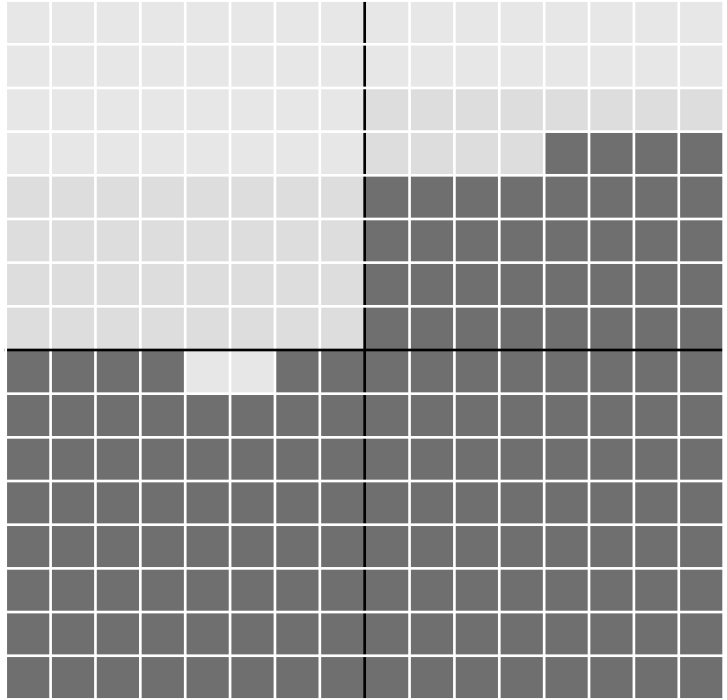
4-Way Mergesort

Algorithmus 4-WAY-MERGE

Eingabe: $k \times k$ Array, dessen $\frac{k}{2} \times \frac{k}{2}$ Teilarrays teilsortiert sind.

Ausgabe: Teilsortiertes $k \times k$ Array

1. Sortiere die Zeilen der *oberen* Teilarrays aufsteigend
2. Sortiere die Zeilen der *unteren* Teilarrays absteigend
3. Sortiere die Spalten
4. Sortiere die Zeilen der *ungeraden* Zeilen aufsteigend
5. Sortiere die Zeilen der *geraden* Zeilen absteigend
6. Sortiere die Spalten



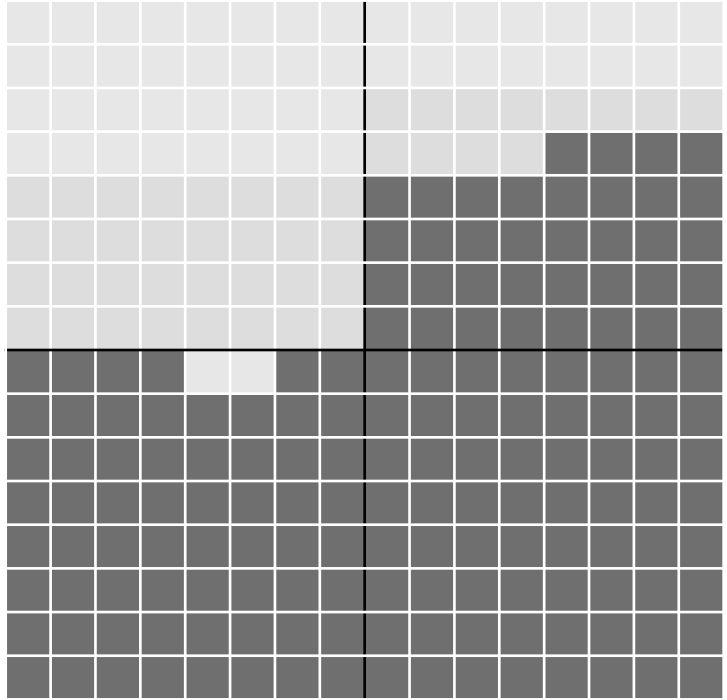
4-Way Mergesort

Algorithmus 4-WAY-MERGE

Eingabe: $k \times k$ Array, dessen $\frac{k}{2} \times \frac{k}{2}$ Teilarrays teilsortiert sind.

Ausgabe: Teilsortiertes $k \times k$ Array

1. Sortiere die Zeilen der *oberen* Teilarrays aufsteigend
2. Sortiere die Zeilen der *unteren* Teilarrays absteigend
3. Sortiere die Spalten
4. Sortiere die Zeilen der *ungeraden* Zeilen aufsteigend
5. Sortiere die Zeilen der *geraden* Zeilen absteigend
6. Sortiere die Spalten



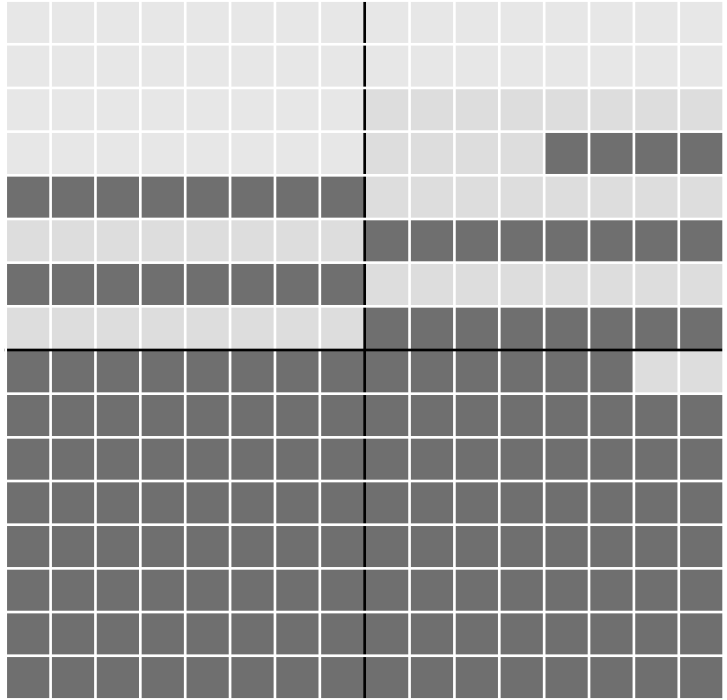
4-Way Mergesort

Algorithmus 4-WAY-MERGE

Eingabe: $k \times k$ Array, dessen $\frac{k}{2} \times \frac{k}{2}$ Teilarrays teilsortiert sind.

Ausgabe: Teilsortiertes $k \times k$ Array

1. Sortiere die Zeilen der *oberen* Teilarrays aufsteigend
2. Sortiere die Zeilen der *unteren* Teilarrays absteigend
3. Sortiere die Spalten
4. Sortiere die Zeilen der *ungeraden* Zeilen aufsteigend
5. Sortiere die Zeilen der *geraden* Zeilen absteigend
6. Sortiere die Spalten



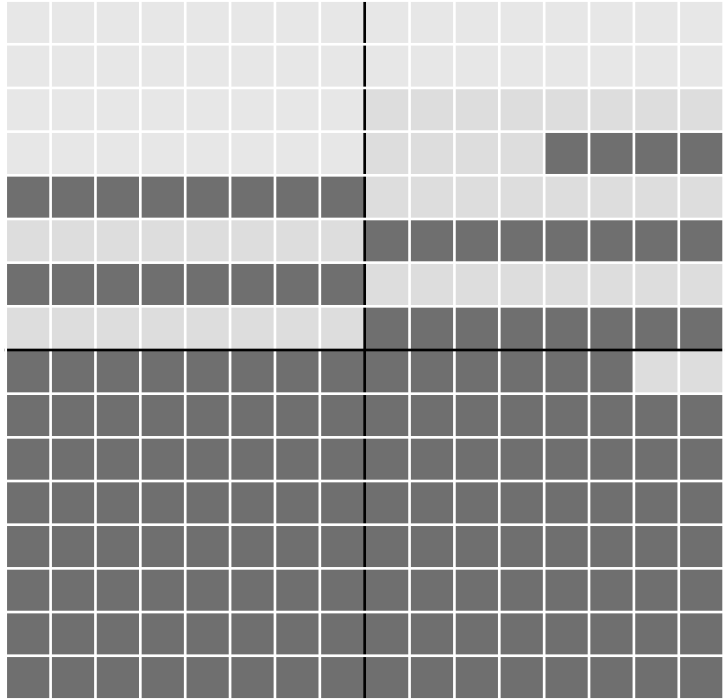
4-Way Mergesort

Algorithmus 4-WAY-MERGE

Eingabe: $k \times k$ Array, dessen $\frac{k}{2} \times \frac{k}{2}$ Teilarrays teilsortiert sind.

Ausgabe: Teilsortiertes $k \times k$ Array

1. Sortiere die Zeilen der *oberen* Teilarrays aufsteigend
2. Sortiere die Zeilen der *unteren* Teilarrays absteigend
3. Sortiere die Spalten
4. Sortiere die Zeilen der *ungeraden* Zeilen aufsteigend
5. Sortiere die Zeilen der *geraden* Zeilen absteigend
6. Sortiere die Spalten



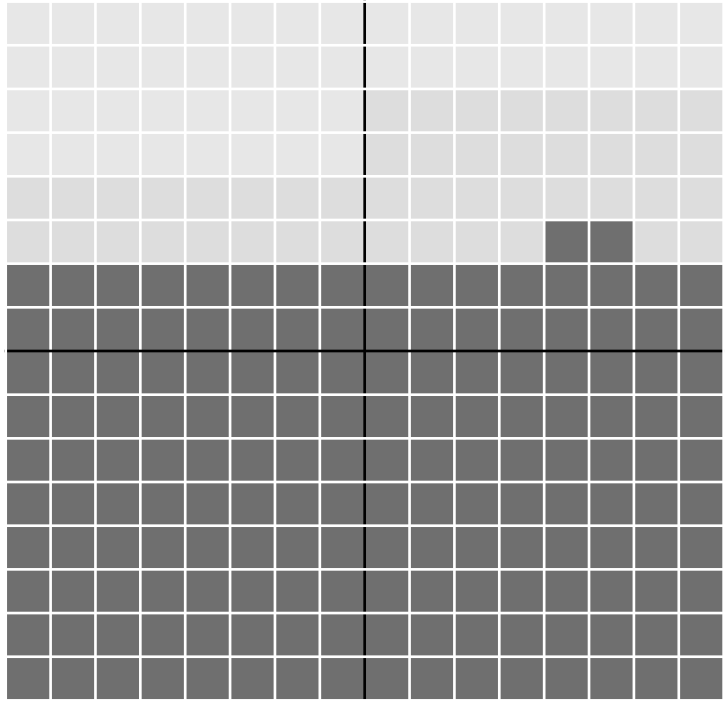
4-Way Mergesort

Algorithmus 4-WAY-MERGE

Eingabe: $k \times k$ Array, dessen $\frac{k}{2} \times \frac{k}{2}$ Teilarrays teilsortiert sind.

Ausgabe: Teilsortiertes $k \times k$ Array

1. Sortiere die Zeilen der *oberen* Teilarrays aufsteigend
2. Sortiere die Zeilen der *unteren* Teilarrays absteigend
3. Sortiere die Spalten
4. Sortiere die Zeilen der *ungeraden* Zeilen aufsteigend
5. Sortiere die Zeilen der *geraden* Zeilen absteigend
6. Sortiere die Spalten



4-Way Mergesort

Algorithmus SPLIT

Eingabe: Unsortiertes $k \times k$ Array a
Ausgabe: Teilsortiertes $k \times k$ Array

1. **if** (*länge*$(a) > 1$) **then**
2. Teile a in 4 gleich große Teilarrays T_i und rufe SPLIT(T_i) für jedes Teilarray auf
3. 4-WAY-MERGE(a)

Algorithmus 4-WAY-MERGESORT

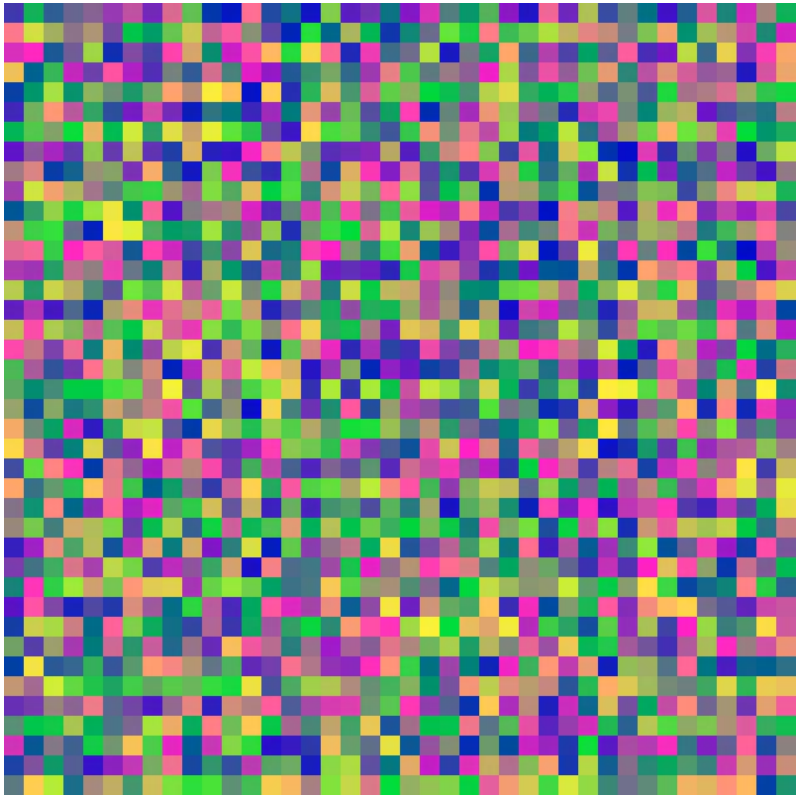
Eingabe: Unsortiertes $n \times n$ Array a
Ausgabe: Sortiertes $n \times n$ Array

1. SPLIT(a)
2. Sortiere die Zeilen

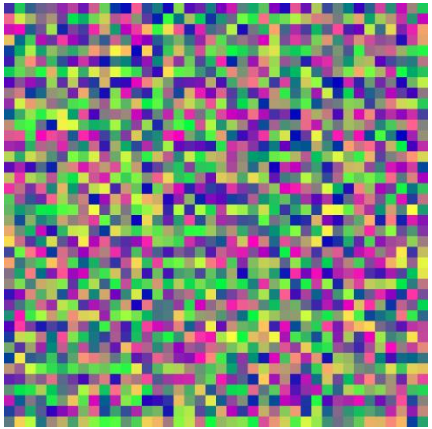
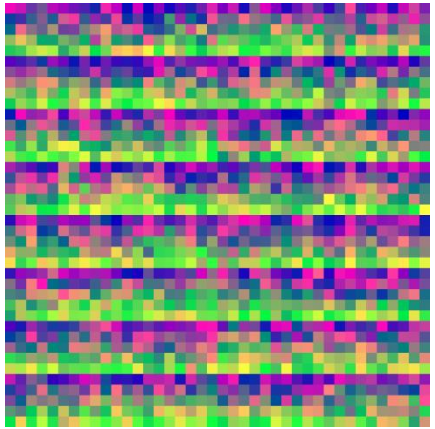
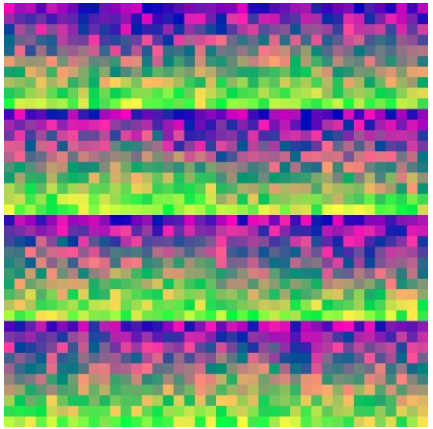
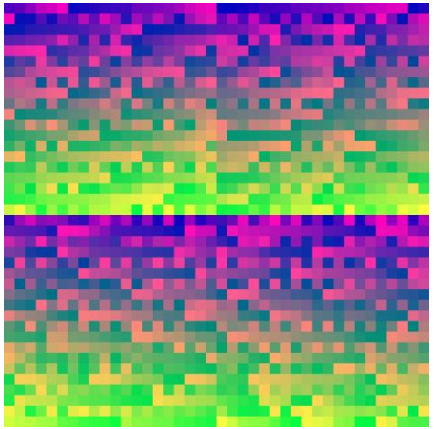
Laufzeit

- Jede Ausführung von 4-WAY-MERGE benötigt $3k$ Schritte. Entsprechend benötigen wir $3n + \frac{3n}{2} + \frac{3n}{4} + \cdots + 3 \leq 6n$ für die Ausführung der Merge-Schritte
- Zusätzlich wird am Ende einmal mit Laufzeit n die Zeilen sortiert.
- $\rightarrow$ Gesamtlaufzeit: $7n$

4-Way Mergesort in Aktion



4-Way Mergesort in Aktion

Alternative: Rotatesort

Idee: Sortiere Array mit Hilfe einer konstanten Anzahl an Zeilen bzw. Spaltenoperationen

Algorithmus ROTATESORT

Eingabe: Unsortiertes $n \times n$ Array a
Ausgabe: Sortiertes $n \times n$ Array

1. Wende BALANCE auf jedem vertikalen Teilarray der Größe $n \times \sqrt{n}$ an
2. UNBLOCK(a)
3. Wende BALANCE auf jedem horizontalem Teilarray der Größe $n \times \sqrt{n}$ an
4. UNBLOCK(a)
5. SHEAR(a)
6. SHEAR(a)
7. SHEAR(a)
8. Sortiere die Zeilen

Algorithmus BALANCE

Eingabe: Teilarray der Form $n \times \sqrt{n}$

1. Sortiere die Spalten
2. Rotiere die Zeilen um $i \bmod \sqrt{n}$
3. Sortiere die Spalten

Algorithmus UNBLOCK

Eingabe: $n \times n$ Array

1. Rotiere die Zeilen um $i \cdot \sqrt{n} \bmod n$
2. Sortiere die Spalten

Algorithmus SHEAR

Eingabe: $n \times n$ Array

1. Sortiere die Zeilen (Alternierend)
2. Sortiere die Spalten

Alternative: Rotatesort

Algorithmus BALANCE

1. Sortiere die Spalten
2. Rotiere die Zeilen um $i \ mod \ \sqrt{n}$
3. Sortiere die Spalten

Algorithmus ROTATESORT

Eingabe: Unsortiertes $n \times n$ Array a
Ausgabe: Sortiertes $n \times n$ Array

1. Wende BALANCE auf jedem vertikalen Teilarray der Größe $n \times \sqrt{n}$ an
2. UNBLOCK(a)
3. Wende BALANCE auf jedem horizontalem Teilarray der Größe $n \times \sqrt{n}$ an
4. UNBLOCK(a)
5. SHEAR(a)
6. SHEAR(a)
7. SHEAR(a)
8. Sortiere die Zeilen

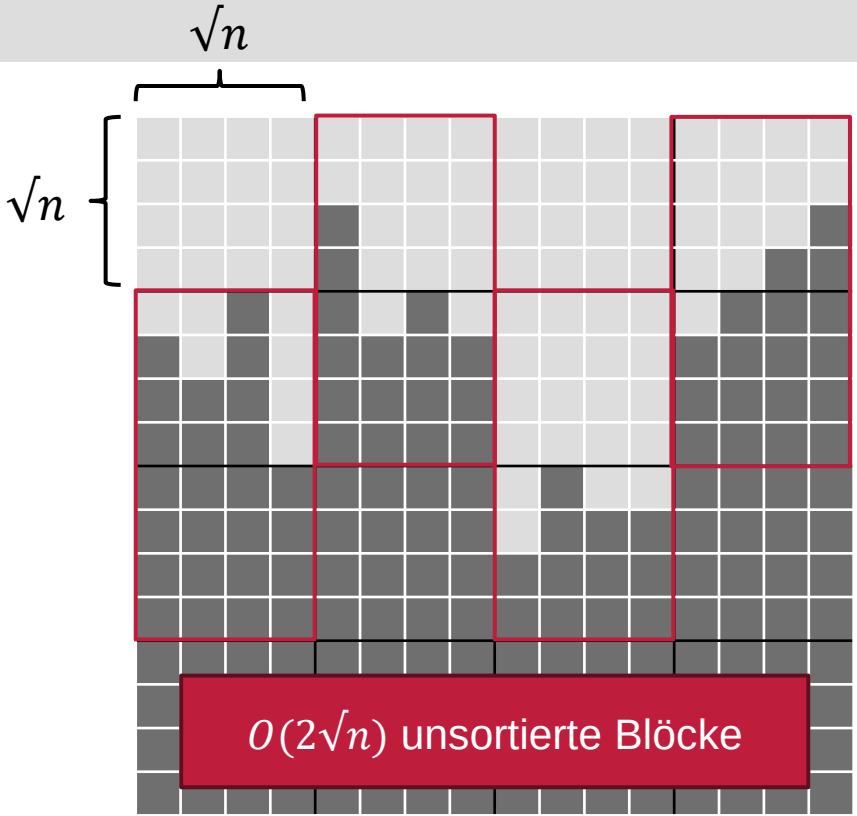
Alternative: Rotatesort

Algorithmus ROTATESORT

Eingabe: Unsortiertes $n \times n$ Array a

Ausgabe: Sortiertes $n \times n$ Array

1. Wende BALANCE auf jedem vertikalen Teilarray der Größe $n \times \sqrt{n}$ an
2. UNBLOCK(a)
3. Wende BALANCE auf jedem horizontalem Teilarray der Größe $n \times \sqrt{n}$ an
4. UNBLOCK(a)
5. SHEAR(a)
6. SHEAR(a)
7. SHEAR(a)
8. Sortiere die Zeilen



Alternative: Rotatesort

Algorithmus ROTATESORT

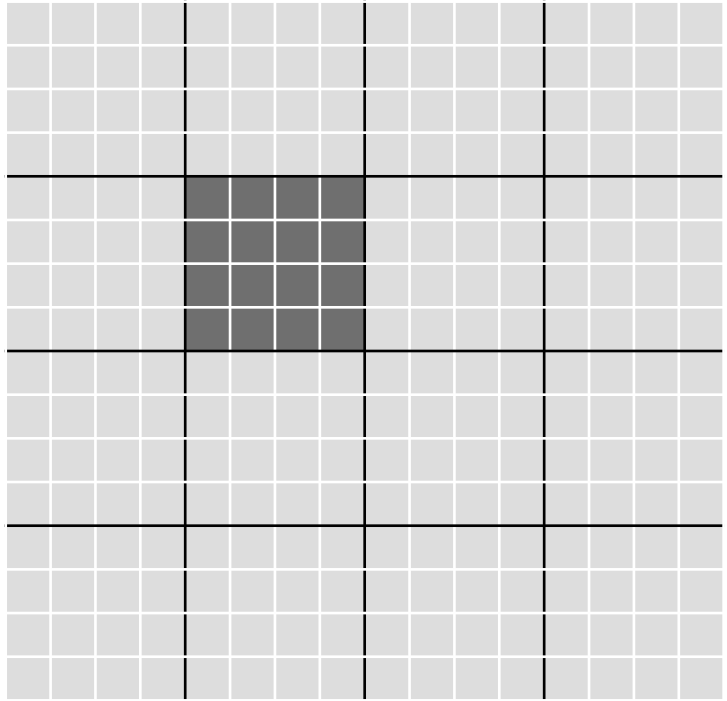
Eingabe: Unsortiertes $n \times n$ Array a
Ausgabe: Sortiertes $n \times n$ Array

1. Wende BALANCE auf jedem vertikalen Teilarray der Größe $n \times \sqrt{n}$ an
2. UNBLOCK(a)
3. Wende BALANCE auf jedem horizontalem Teilarray der Größe $n \times \sqrt{n}$ an
4. UNBLOCK(a)
5. SHEAR(a)
6. SHEAR(a)
7. SHEAR(a)
8. Sortiere die Zeilen

Algorithmus UNBLOCK

Eingabe: $n \times n$ Array

1. Rotiere die Zeilen $i \cdot \sqrt{n} \bmod n$
2. Sortiere die Spalten



Alternative: Rotatesort

Algorithmus ROTATESORT

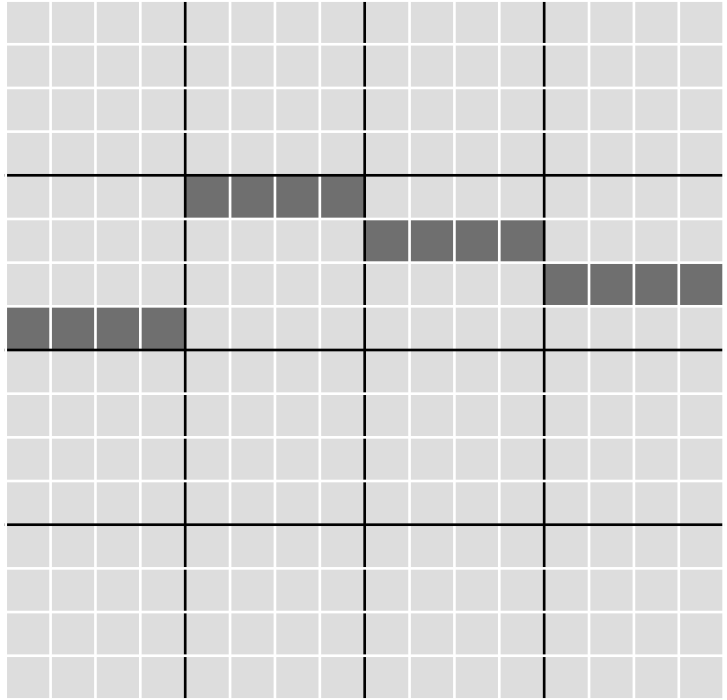
Eingabe: Unsortiertes $n \times n$ Array a
Ausgabe: Sortiertes $n \times n$ Array

1. Wende BALANCE auf jedem vertikalen Teilarray der Größe $n \times \sqrt{n}$ an
2. UNBLOCK(a)
3. Wende BALANCE auf jedem horizontalem Teilarray der Größe $n \times \sqrt{n}$ an
4. UNBLOCK(a)
5. SHEAR(a)
6. SHEAR(a)
7. SHEAR(a)
8. Sortiere die Zeilen

Algorithmus UNBLOCK

Eingabe: $n \times n$ Array

1. Rotiere die Zeilen $i \cdot \sqrt{n} \bmod n$
2. Sortiere die Spalten



Alternative: Rotatesort

Algorithmus ROTATESORT

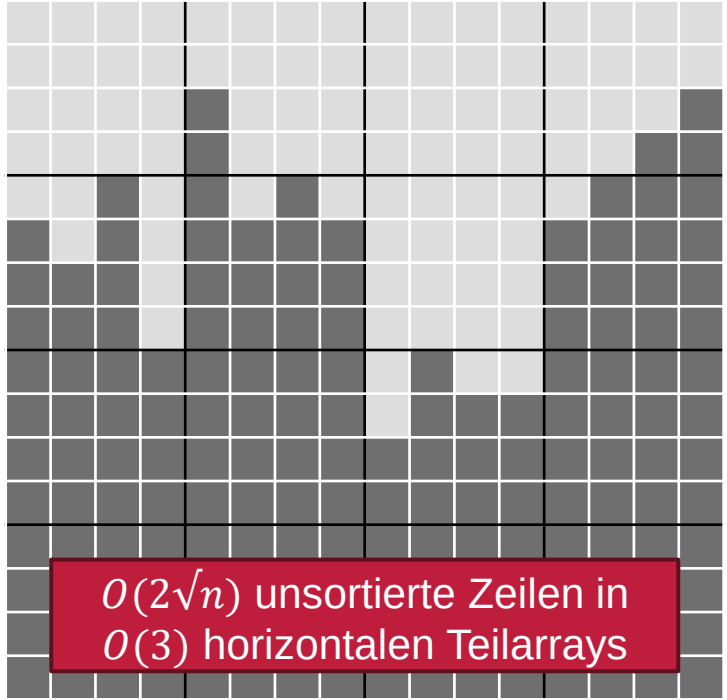
Eingabe: Unsortiertes $n \times n$ Array a
Ausgabe: Sortiertes $n \times n$ Array

1. Wende BALANCE auf jedem vertikalen Teilarray der Größe $n \times \sqrt{n}$ an
2. UNBLOCK(a)
3. Wende BALANCE auf jedem horizontalem Teilarray der Größe $n \times \sqrt{n}$ an
4. UNBLOCK(a)
5. SHEAR(a)
6. SHEAR(a)
7. SHEAR(a)
8. Sortiere die Zeilen

Algorithmus UNBLOCK

Eingabe: $n \times n$ Array

1. Rotiere die Zeilen $i \cdot \sqrt{n} \bmod n$
2. Sortiere die Spalten



Alternative: Rotatesort

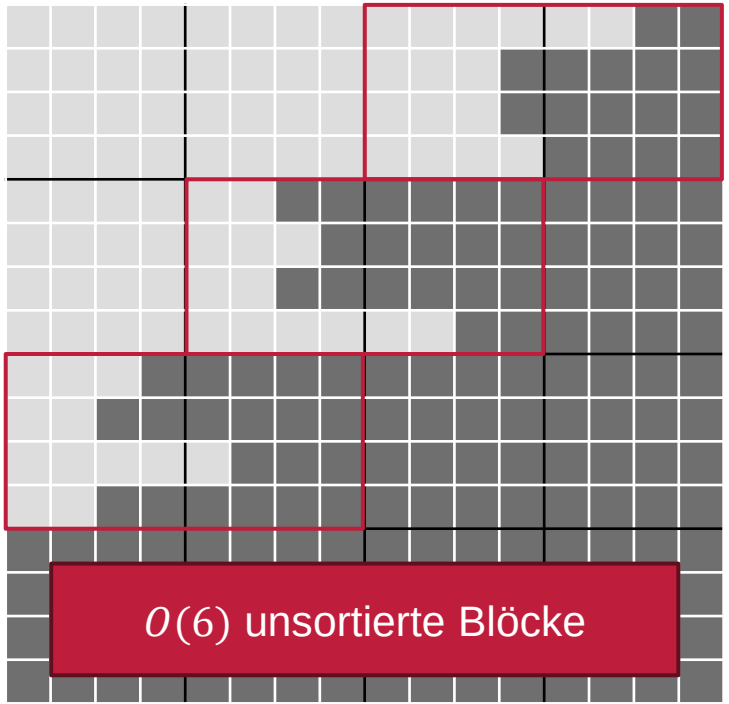
Algorithmus ROTATESORT

Eingabe: Unsortiertes $n \times n$ Array a
Ausgabe: Sortiertes $n \times n$ Array

1. Wende BALANCE auf jedem vertikalen Teilarray der Größe $n \times \sqrt{n}$ an
2. UNBLOCK(a)
3. Wende BALANCE auf jedem horizontalem Teilarray der Größe $n \times \sqrt{n}$ an
4. UNBLOCK(a)
5. SHEAR(a)
6. SHEAR(a)
7. SHEAR(a)
8. Sortiere die Zeilen

Algorithmus BALANCE

1. Sortiere die Spalten
2. Rotiere die Zeilen um $i \bmod \sqrt{n}$
3. Sortiere die Spalten



Alternative: Rotatesort

Algorithmus ROTATESORT

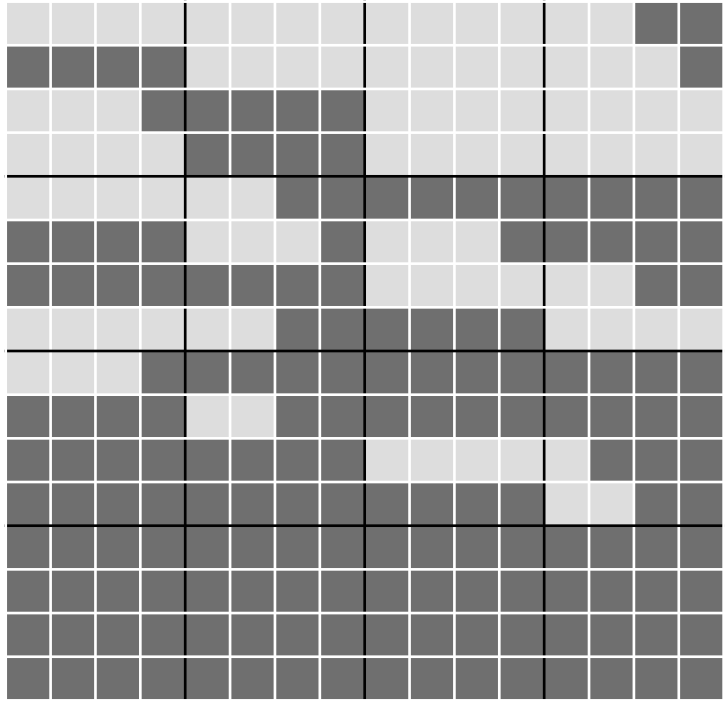
Eingabe: Unsortiertes $n \times n$ Array a
Ausgabe: Sortiertes $n \times n$ Array

1. Wende BALANCE auf jedem vertikalen Teilarray der Größe $n \times \sqrt{n}$ an
2. UNBLOCK(a)
3. Wende BALANCE auf jedem horizontalem Teilarray der Größe $n \times \sqrt{n}$ an
4. UNBLOCK(a)
5. SHEAR(a)
6. SHEAR(a)
7. SHEAR(a)
8. Sortiere die Zeilen

Algorithmus UNBLOCK

Eingabe: $n \times n$ Array

1. Rotiere die Zeilen $i \cdot \sqrt{n} \bmod n$
2. Sortiere die Spalten



Alternative: Rotatesort

Algorithmus ROTATESORT

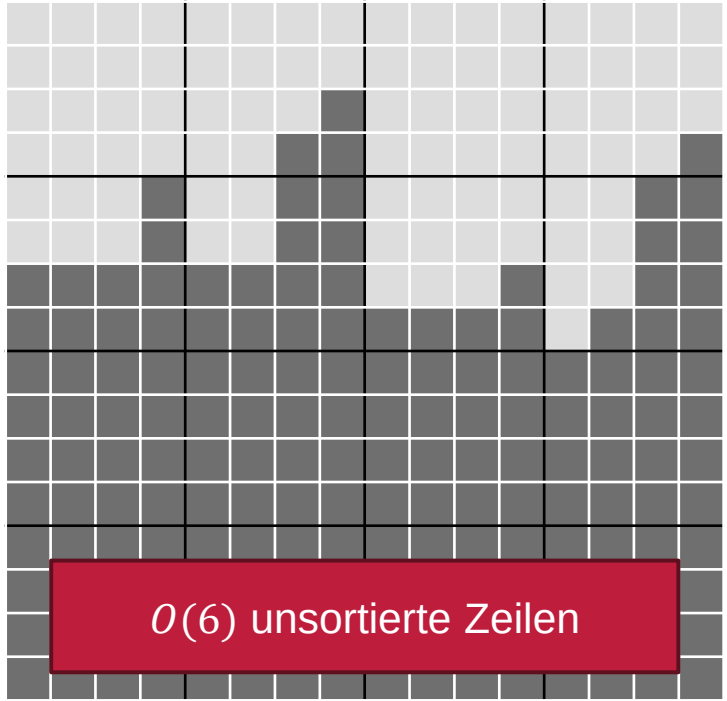
Eingabe: Unsortiertes $n \times n$ Array a
Ausgabe: Sortiertes $n \times n$ Array

1. Wende BALANCE auf jedem vertikalen Teilarray der Größe $n \times \sqrt{n}$ an
2. UNBLOCK(a)
3. Wende BALANCE auf jedem horizontalem Teilarray der Größe $n \times \sqrt{n}$ an
4. UNBLOCK(a)
5. SHEAR(a)
6. SHEAR(a)
7. SHEAR(a)
8. Sortiere die Zeilen

Algorithmus UNBLOCK

Eingabe: $n \times n$ Array

1. Rotiere die Zeilen $i \cdot \sqrt{n} \bmod n$
2. Sortiere die Spalten



Alternative: Rotatesort

Algorithmus ROTATESORT

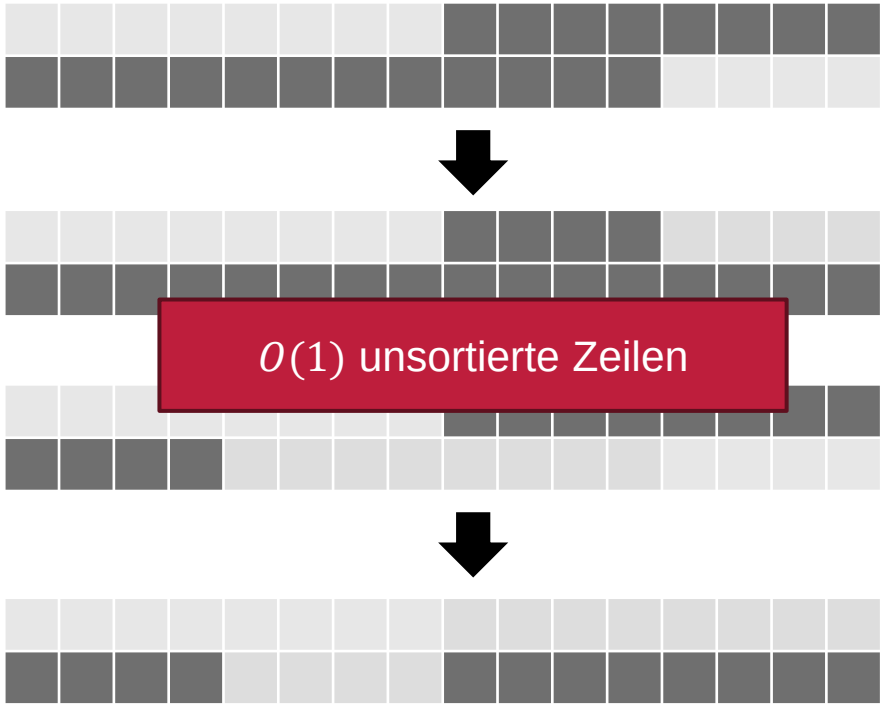
Eingabe: Unsortiertes $n \times n$ Array a
Ausgabe: Sortiertes $n \times n$ Array

1. Wende BALANCE auf jedem vertikalen Teilarray der Größe $n \times \sqrt{n}$ an
2. UNBLOCK(a)
3. Wende BALANCE auf jedem horizontalem Teilarray der Größe $n \times \sqrt{n}$ an
4. UNBLOCK(a)
5. SHEAR(a)
6. SHEAR(a)
7. SHEAR(a)
8. Sortiere die Zeilen

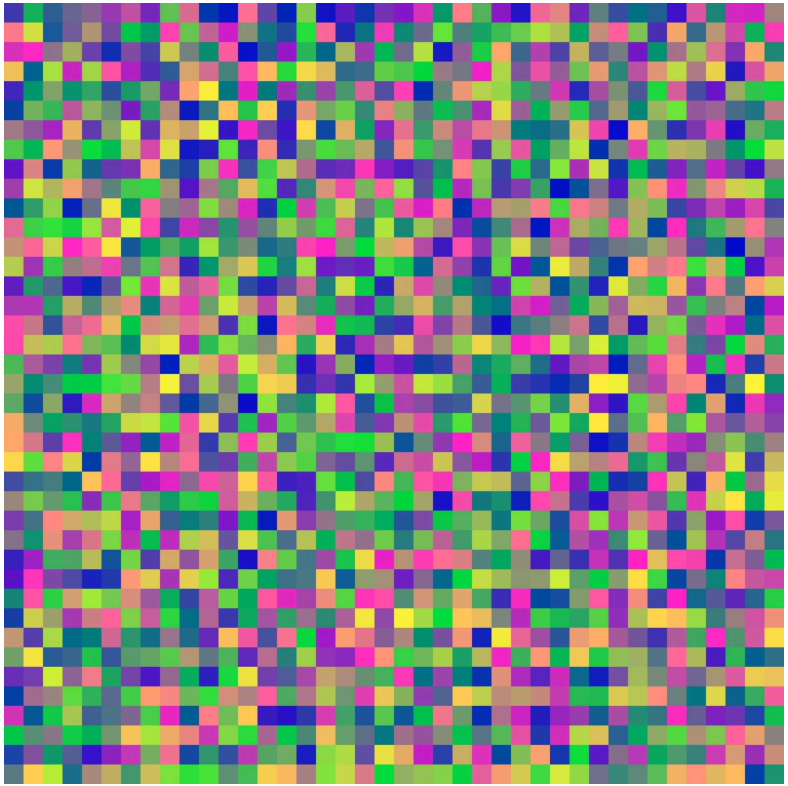
Algorithmus SHEAR

Eingabe: $n \times n$ Array

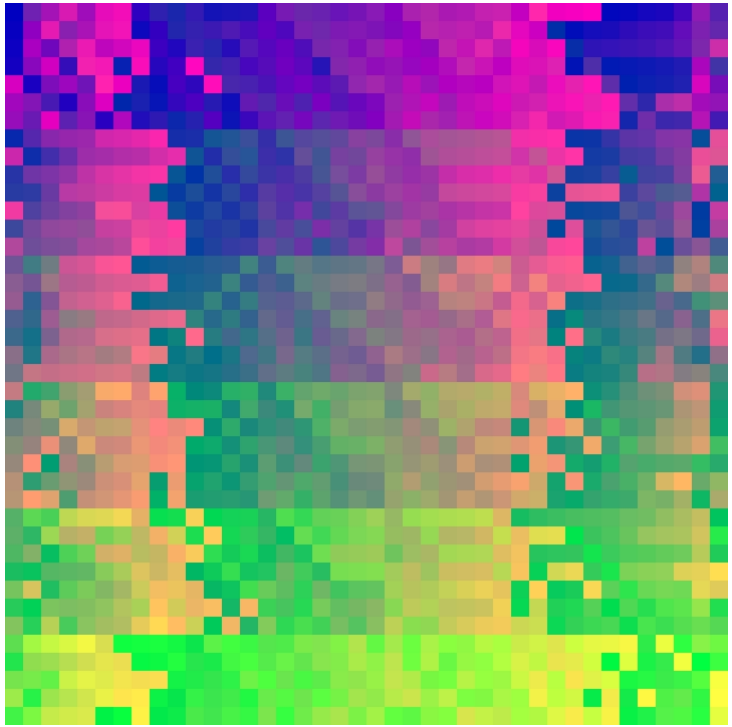
1. Sortiere die Zeilen (Alternierend)
2. Sortiere die Spalten

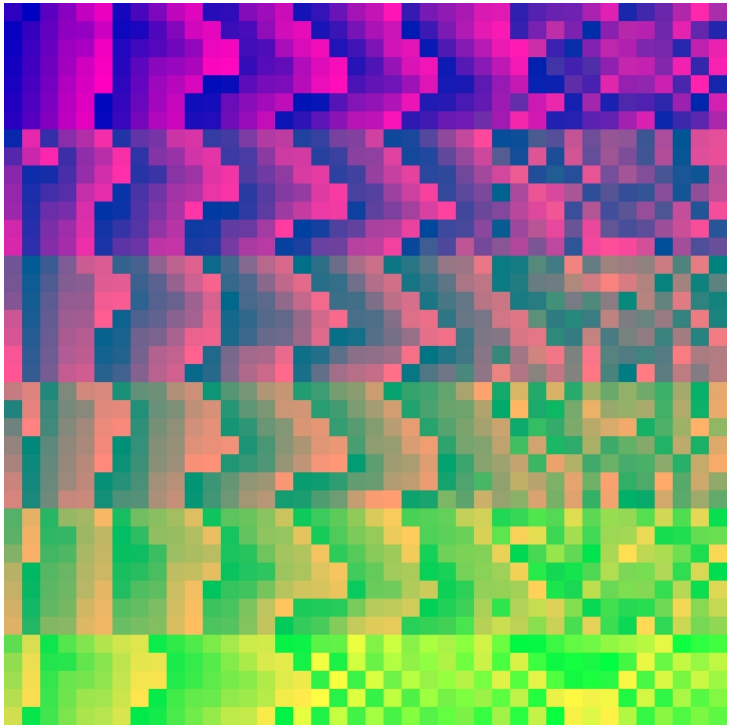


Rotatesort in Aktion



Rotatesort in Aktion





Obere Schranke

An Optimal Sorting Algorithm
For Mesh Connected Computers

C.P. Schnorr
Mathematics Dept and Computer Science Dept,
University of Frankfurt, W. Germany

A. Shamir
Applied Mathematics Dept,
The Weizmann Institute, Israel

Abstract. In this paper we prove a $3n$ upper and lower bound on the complexity of sorting on a $n \times n$ mesh connected parallel computer, and describe an exceptionally simple algorithm which sorts the array by alternately sorting its rows and columns $\log \log n$ times.

1. Introduction

Sorting is a fundamental problem with a rich theory and important practical applications. Its sequential complexity is fairly well understood, with $0(n \log n)$ upper and lower bounds in most machine models. The recent discovery of the Ajtai, Komlos and Szemeredi [1983] sorting network and the subsequent works of Bilardy and Preparata [1985] and of Leighton [1985] established $0(\log n)$ as the parallel complexity of sorting in certain machine models. However, the high wire density and the enormous constant associated with the AKS network make it highly desirable to find simple parallel sorting algorithms even if their asymptotic complexity is suboptimal.

In this paper we consider one of the simplest models of parallel computers – the two-dimensional mesh connected computer. While it is usually slower than machines based on the hypercube, the cube connected cycles, or the shuffle exchange graphs, it is ideally suited to VLSI implementation. The locality of the communication, the of the leading term but ignore the low order additive terms in complexity functions). Subsequent algorithms published by Nassimi and Sahni [1979] and by Kumar and Hirschberg [1983], sort a $n \times n$ array in $14n$ and $11n$ steps, respectively, but they have a better behaviour for small n. Recently, Lang Schimmler Schmeck and Schroder [1985] published a $7n$ algorithm which is slightly easier to implement, and Leighton discovered (but did not publish) a $4n$ algorithm.

The only lower bound known for this problem is the trivial bound based on distance: to move an element from the lower right corner to the upper left corner requires at least $2n-2$ steps (some papers quote a $4n-4$ lower bound, but this is just the distance bound in a more restricted SIMD model). While the distance lower bound was known to be tight for one dimensional arrays, the real complexity of two dimensional sorting remained an open problem.

In this paper we increase the lower bound and decrease the upper bound on the complexity of sorting a $n \times n$ input into row-major order on a mesh connected computer to $3n$. The lower bound is proven for a very strong model, while the upper bound is based on a very weak model, and thus the new algorithm is proven optimal (up to low order terms) for a wide variety of models. The proof of optimality can be extended to rectangular arrays with r rows and c columns (as long as $c \leq r^2$), since both the upper bound and lower bound in this case

Obere und untere Schranke: $3n$

Verwandte Probleme

0	8	2	3	11	5	6
14	1	9	17	4	12	13
7	15	16	10	18	19	20
21	22	23	24	25	26	27
28	36	30	32	31	33	41
35	29	37	38	39	40	34
42	43	44	45	46	47	48

0	1	2
3	4	5