Skript Rekonfiguriere Rechensysteme (Ergänzungen)

Martin Middendorf

Universität Leipzig

Proof for DAG Model

Idea: Use Dynamic Programming and compute a table

$$M = (M_{k,j})_{k \in [1:m], j \in [k:m]}$$

where $M_{k,j}$ are the minimal costs for the prefix of length j of the sequence of context requirements $c_1 \ldots c_m$ when using k hypercontexts.

$\rightarrow$ the optimal solution can be derived with standard dynamic programming techniques from M

Let h_{ij} be a cheapest hypercontext that satisfies contexts requirements $c_i, \ldots, c_j$

Algorithm PHC-DAG:

i. Preprocessing: For each $i, j \in [1 : m]$, $i < j$ cost $cost(h_{ij})$ is computed

ii. Initialization: Every element in first row of M is determined, i.e., for $j \in [1 : m]$ compute

$$M_{1,j} := w + cost(h_{1j}) \cdot j$$

iii. Computation of $M_{k,j}$ for $k \in [2 : m], j \in [k : m]$ according to

$$M_{k,j} = min\{M_{k-1,p-1} + w + cost(h_{p,j}) \cdot (j - p - 1) \mid p \in [k : j]\}$$

iv. Computation of the quality of the optimal solution from M by determining

$$\min\{M_{k,p} \mid k \in [1 : m]\}$$

Run time analysis:

Preprocessing step takes time $O(m^2 \cdot \alpha)$

Initialization takes time $O(m \cdot \alpha)$ since each element can be determined in time $O(\alpha)$.

For (iii) consider an element $M_{k,j}$ with $k > 1$ and assume that all elements in row $k-1$ have already been determined. Then it is clear that the computation of a single element in (iii) takes time at most time $O(j)$.

Step (iv) takes time $O(m)$.

Cost Model for Multi Task Hyperreconfigurable Architectures

Let

- H^{pub} be the set of **public global resources**
- H^{priv} be the set of **private global resources**
- H^{loc} be the set of **local resources**
- $\mathcal{H}$ be the set of **global hypercontexts**
- $\mathcal{H}^{loc}$ the set of **local hypercontexts**
- $\mathcal{H}^{loc,priv}$ be the set of **extended local hypercontexts**

Each **global hypercontext** $h \in \mathcal{H}$ is a vector $(h_0, h_1, \ldots, h_m)$ with $h_0 \subset H^{pub}$ and $h_i \subset H^{priv}$, $i \in [1 : m]$ where h_0 defines the available public global resources and h_j defines the assignment of private global resources to task T_j, $j \in [1 : m]$.
If public global resources do not exist h has the form $(h_1, \ldots, h_m)$ with $h_i \subset H^{priv}$, $i \in [1 : m]$.

Similarly, each **extended local hypercontext** $h^{loc,priv} \in \mathcal{H}^{loc,priv}$ is a vector $((h_1^{loc}, h_1^{priv}), \ldots, (h_m^{loc}, h_m^{priv}))$ with $h_i^{loc} \in H^{loc}$ and $h_i^{priv} \subset H^{priv}$, $i \in [1 : m]$ where h_i^{loc} defines the available local resources and h_j^{priv} the available private global resources that are owned by task T_j, $j \in [1 : m]$ (and therefore can be changed in a local hyperreconfiguration by task T_j).

For a given hypercontext $(h_0, h_1, \ldots, h_m)$ and a given (fixed) assignment $(f_1^{loc}, \ldots, f_m^{loc}) \in (\mathcal{H}^{loc})^m$ of local resources to the tasks an extended local hypercontext $((h_1^{loc}, h_1^{priv}), \ldots, (h_m^{loc}, h_m^{priv}))$ is **valid** only if $h_j^{priv} \subset h_j$ and $h_j^{loc} \subset f_j^{loc}$ for all $j \in [1:m]$.

For a global hypercontext $h \in \mathcal{H}$ let $\mathcal{H}_h^{loc,priv}$ be the set of extended local hypercontexts that are valid under h.

How the **costs** for performing a hyperreconfiguration are counted depends on whether this is done task parallel or task sequentially and synchronized or asynchronous.

Assume: m tasks $T_1, \ldots, T_m$ run on the machine, $(f_1^{loc}, \ldots, f_m^{loc}) \in (\mathcal{H}^{loc})^m$ is the assignment of local resources to tasks and T_j, $j \in [1:m]$ executes between global hyperreconfiguration $h = (h_0, \ldots, h_m)$ and the next global hyperreconfiguration h' a sequence $(h_{j,1}^{loc}, h_{j,1}^{priv})S_{j,1} \ldots (h_{j,n_j}^{loc}, h_{j,n_j}^{loc})S_{j,n_j}$, $n_j \geq 1$ of valid local hyperreconfiguration and reconfiguration operations where $(h_{j,i}^{loc}, h_{j,i}^{priv}) \in \mathcal{H}_h^{loc,priv}$ is a local hyperreconfiguration and $S_{j,i}$ is a sequence of context requirements, $i \in [1:n_j]$.

Asynchronous Case: where reconfigurations and partial hyperreconfigurations are executed task parallel

i. Multi Task Switch (MT-Switch) model:

Given

- set of **public global reconfigurable units** (or switches) X^{pub}
- set of **private global reconfigurable units** (or switches) $X^{priv} = \{x_1, \ldots, x_u\}$
- set of **local reconfigurable units** (switches) $X^{loc} = \{z_1, \ldots, z_v\}$

Then $H^{priv} = X^{priv}$ and $H^{loc} = X^{loc}$, $(f_1^{loc}, \ldots, f_m^{loc})$ is a partition of a subset of X^{loc} and a local hypercontext is a partition of a subset of X^{priv}

- **Cost function**
  - $cost(h^{loc}, h^{priv}) = |h^{loc}| + |h^{priv}|$ where $h^{loc} \subset X^{loc}$ and $h^{priv} \subset X^{priv}$ and
  - $init(h) = w > 0$ for $h \in \mathcal{H}$
    Typical special case: $w = |X^{pub}| + |X^{priv}|$
  - $init(h_j, f_j^{loc}) = v_j > 0$ for $h_j \in \mathcal{H}^{priv}$, $f_j^{loc} \in \mathcal{H}^{loc}$
    Typical special case: $v_j = |h_j| + |f_j^{loc}|$.

The maximal total (hyper)reconfiguration time of all tasks from h to h' is

$$w + \max_{j=1}^{m}\{\sum_{i=1}^{n_j}(v_j + (|h_{i,j}^{loc}| + |h_{j,i}^{priv}|) \cdot |S_{j,i}|)\}$$

and a typical special case is

$$|X^{pub}| + |X^{priv}| + \max_{j=1}^{m}\{\sum_{i=1}^{n_j}(|h_j| + |f_j^{loc}| + (|h_{i,j}^{loc}| + |h_{i,j}^{priv}|) \cdot |S_{j,i}|)\}$$

Proof for Switch Model

Idea: Use dynamic programming and compute table $M = (M_{k,j})_{k\in[1:m],j\in[k:m]}$ where $M_{k,j}$ are the minimal costs for the prefix of length j of the sequence of context requirements $c_1 \dots c_m$ when using k hypercontexts

$\rightarrow$ The optimal solution for PHC-Switch can then be derived from this matrix

Idea: Algorithm is designed such that each row of M can be determined in time $O(n \cdot m) \Rightarrow$ total run time is $O(n \cdot m^2)$

Let h_{ij} be a cheapest hypercontext that satisfies the contexts requirements $c_i, \dots, c_j$.

Inhaltsverzeichnis (vorläufig)

Einführung
1. Berechnungsmodelle I
2. Vergleich der Berechnungsmodelle I
3. Grundlegende Algorithmen I

    3.1 logisches ODER
    3.2 XOR, Parität

4. Vergleich der Berechnungsmodelle II
5. Grundlegende Algorithmen II

    5.1 Kompaktifizieren
    5.2 Summe einer Folge von Bits
    5.3 Summe einer Folge binärer Zahlen
    5.4 Sortieren

Rekonfigurierbare Gitter

Literatur:

Bücher:

1. T.H Cormen, C.E. Leiserson, R.L. Rivest: *Introduction to Algorithms*. MIT Press, Cambridge, MA, (1990)

2. H. Li, Q.F. Stout (Hrsg.): *Reconfigurable massively parallel computers*. Prentice Hall, Englewoord Cliffs, NJ, 1991

3. H. Schmeck: *Analyse von VLSI-Algorithmen*. Spektrum Akademischer Verlag, Heidelberg, 1995.

Aufsätze:

1. B. Beresford-Smith, O. Diessel, H. ElGindy: Optimal algorithms for constrained reconfigurable meshes. *Technical Report*, University of Newcastle, Australien, 1996

2. V. Bokka, H. Gurla, S. Olariu, J.L. Schwing: Constant-time convexity problems on reconfigurable meshes. *J. Par. Distr. Comput.*, 27:86-99

(1995)

3. J.M. Marberg, E. Gafni: Sorting on constant nymber of row and column phases on a mesh. *Algorithmica*, 3:561-572, (1988)

4. J.-W. Jang, H. Park, V.K. Prasanna: A bit model of reconfigurable mesh. *IPPS 94 Workshop on Reconfigurable Architektures*, 22 S. (1994)

5. H. Li, Q.F. Stout: Reconfigurable SIMD massively parallel computers. *Proc. of the IEEE*, 79:429-443 (1991)

6. R. Lin, S. Olariu, J.L. Schwing, J. Zhang: Simulating enhenced meshes with applications. *Par. Proc. Lett.*, 3:59-70 (1993)

7. P.D. MacKenzie: A separation between reconfigurable mesh models. *Par. Proc. Lett.*, 5:15-22 (1995)

8. R. Miller, V.K. Prasanna-Kumar, D.I. Reisis, Q.F. Stout: Parallel computations on reconfigurable mesh. *IEEE Trans. Comput.*, 42: 678-692 (1993)

9. K. Nakano: Efficient summing algorithms for a reconfigurable mesh. *Proc. Workshop on Reconf. Arch.*, Cancun, Mexico (1994)

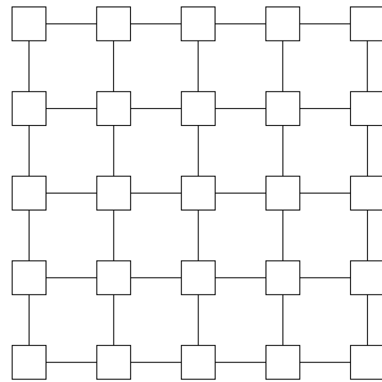
10. M. Nigam, S. Sahni: Sorting n numbers on $n \times n$ reconfigurable meshes with busses. *J. Par. Distr. Comput* 23:37-48 (1995)

11. S. Olariu, J.L. Schwing, J. Zhang: Optimal convex hull algorithms on enhanced meshes. *BIT*, 33:396-410 (1993)

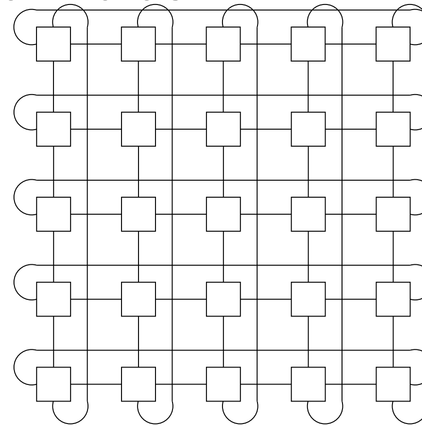
12. H. Schmeck, H. Schröder, G. Turner: Efficient matrix multiplication on a reconfigurable mesh. *Bericht 328*, Institut AIFB, Universität Karlsruhe (1995)

Einführung

Wichtige Verbindungsstrukturen für Parallelrechner mit einer großen Anzahl von Prozessoren sind das Gitter und der Torus.



Gitter (Mesh)



Torus

Vorteile dieser Strukturen sind:

- regulärer Aufbau
- konstanter (niedriger) Knotengrad
- kurze Leitungslängen (Torus bei entsprechender Faltung)
- relativ gut in der Größe veränderbar
- VLSI gerecht
- natürliche Struktur für bestimmte Anwendungen: z.B. Bildverarbeitung

Nachteile dieser Strukturen sind:

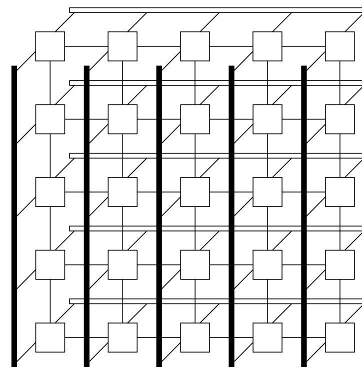
- großer Durchmesser: $2n - 2$ im $n \times n$ Gitter, $2\lfloor \frac{n}{2} \rfloor$ im $n \times n$ Torus
- Broadcast Operationen erforden viel Zeit

Für die Bewertung von Algorithmen spielen Annahmen über die Laufzeit $\sigma(l)$ eines Signals über eine Leitung der Länge l eine wichtige Rolle:

- $\sigma(l) = \Theta(l^2)$ (**Quadratisches Modell**)
- $\sigma(l) = \Theta(l)$ (**Lineares Modell**)
- $\sigma(l) = \Theta(\log l)$ (**Logarithmisches Modell**)
- $\sigma(l) = \Theta(1)$ (**Konstantes Modell**)

In dieser Vorlesung werden wir das Konstante Modell oder — seltener — das Logarithmische Modell zugrundelegen.

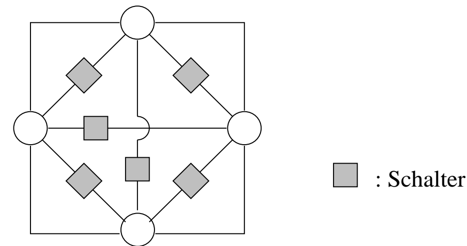
Um die Nachteile des (einfachen) Gitters zu beheben wurde vorgeschlagen dieses mit Zeilen- und Spaltenbussen zu ergänzen:



Gitter mit Zeilen-/Spaltenbussen

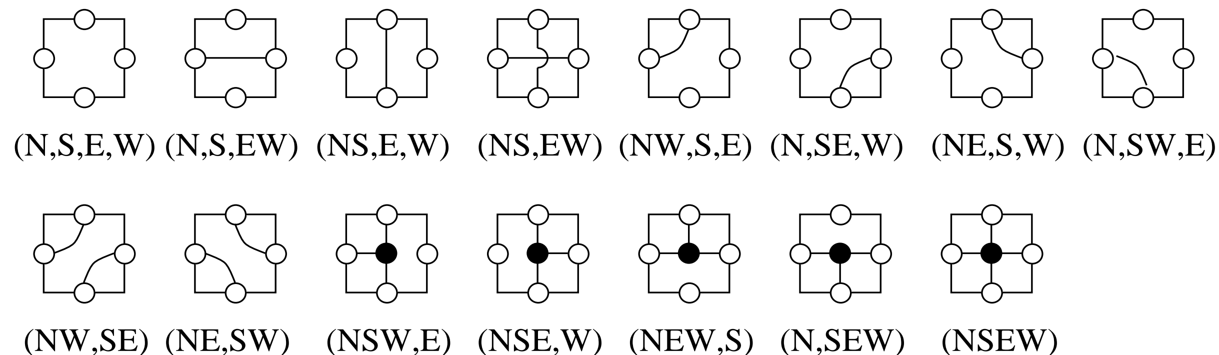
Problem: Häufig möchte man andere Busse schalten können, als Zeilen- oder Spaltenbusse.

Vorschlag : Jeder Prozessor eines Gitters kann Teilmengen seiner 4 Ports lokal durch Schalter verbinden.



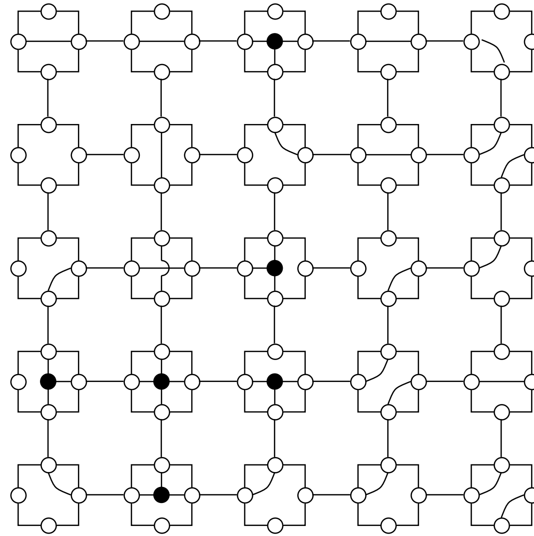
Prozessor mit Schaltern

Es ergeben sich folgende Verbindungsmöglichkeiten in einem Prozessor:

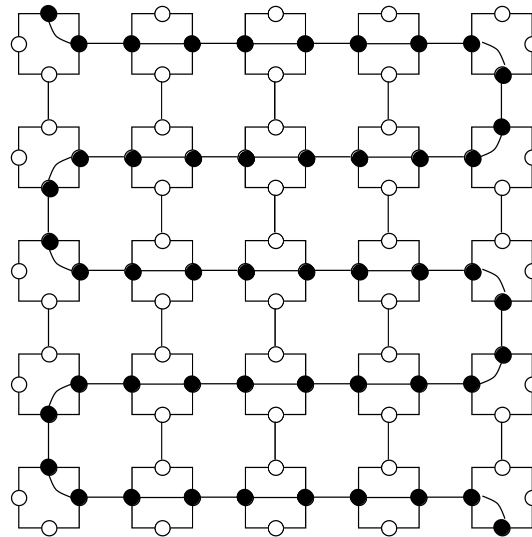


Mögliche Verbindungen

Beispiel für ein konfiguriertes Gitter:



Durch Konfiguration in Schlangenlinie ensteht ein globaler Bus:



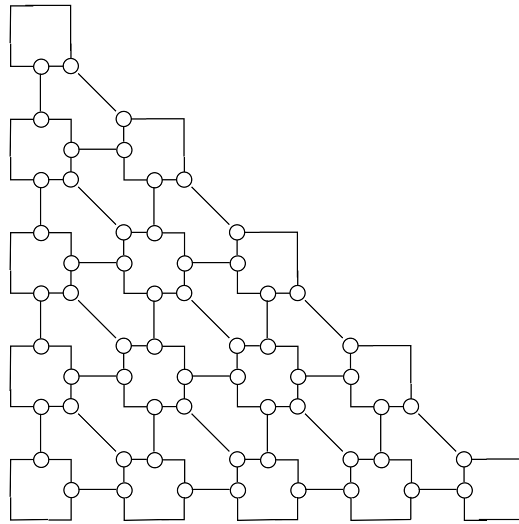
Konfiguration in Schlangenlinie

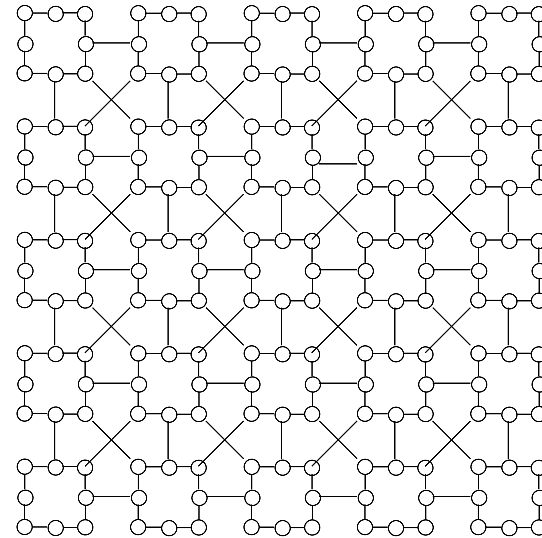
Sprechweise: Die gekennzeichneten Ports seien die Ports entlang der Schlangenlinie. Der nördliche Port von $P_{1,1}$ ist der Anfang der Schlangenlinie.

Problem: Signal auf langem Kanal muß über viele Schalter laufen

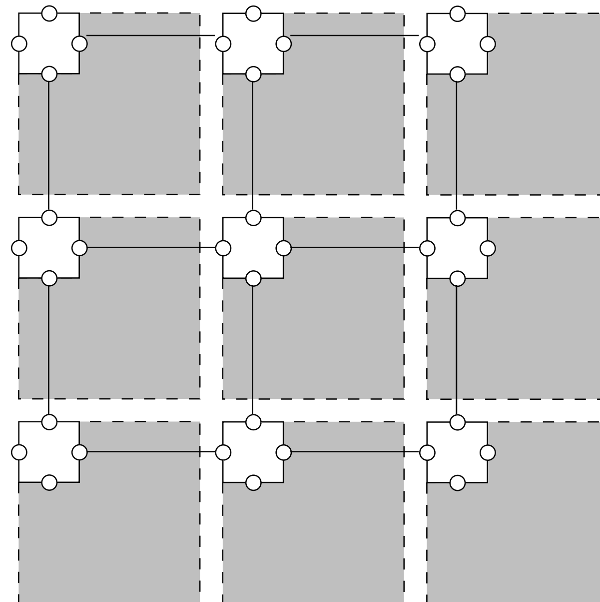
Vorschlag: Uhr mit variabler Takzeit einführen — Taktzeit abhängig von der Länge des längsten Busses. Dies setzt jedoch voraus, daß diese Länge bekannt ist (oft nicht der Fall).

Alternative Topologien





Prozessorfeld mit rekonfigurierbarem Teilgitter



Rekonfigurierbare Prozessorfelder

Ein **Rekonfigurierbares Prozessorfeld** besteht aus einer Menge von m Prozessoren (Prozessor Elementen, **PE's**), die über ein Verbindungsnetz miteinander verbunden sind.

Im **Verbindungsnetz** werden jeweils zwei PE's über eine Leitung miteinander verbunden. Die PE's besitzen jeweils nur konstant viele Nachbarn.
In der Vorlesung ist das Verbindungsnetz meist ein Gitter: **Rekonfigurierbares Gitter** (**RG**).

Eine Leitung besteht aus w vielen Bit-Leitungen.
Standardannahmen: $w = O(\log m)$ (**Wort-Modell**), $w = 1$ (**Bit-Modell**).

Jedes PE kennt seine eindeutige Identifikationsnummer (**PID**).

Jeder Prozessor besitzt **lokalen Speicher**, der $O(1)$ Worte speichern kann.
:Länge der Speicherworte: $O(\log m)$ (Standardannahme im Wort-Modell).

Die PE's arbeiten **synchron** nach dem **SIMD**-Prinzip.

Schreiben/Lesen auf den Bussen (meist): **CREW**-Modell.

> Problem: Was liest ein PE auf einem Bus, auf den kein PE schreibt?
> Wir werden meist annehmen, daß eine 0 gelesen wird.

Pro Takt führt jedes PE folgende Schritte aus:

1. **Rekonfigurierung**: Teilmengen der eigenen Ports werden zusammengeschaltet. Die Entscheidung welche Teilmenge dies ist kann abhängen von:

   – der PID und dem Takt oder

   – der PID, dem Takt und den lokalen Daten (Standardannahme)

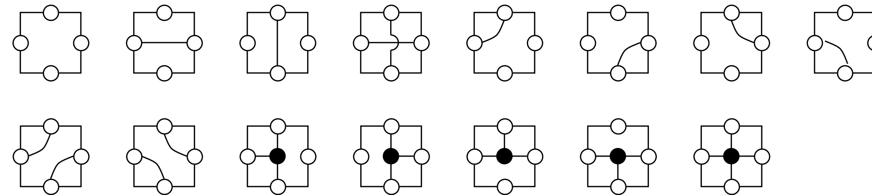
2. Lesen oder Schreiben von bzw. auf einen Port sowie Lesen im lokalen Speicher

3. Ausführen einer arithmetischen/logischen Operation

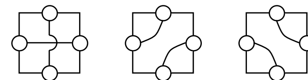
4. Schreiben in den lokalen Speicher

Klassifikation nach den möglichen lokalen Verbindungen:

Beim (allgemeinen) **Rekonfigurierbaren Gitter** (**RG**) dürfen beliebige Ports in einem PE miteinander verbunden werden:

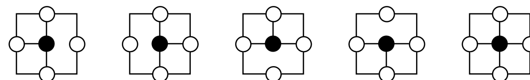


Beim **1-RG** darf ein PE höchstens eine Teilmenge seiner Ports zusammenschalten, d.h. nicht (!) möglich sind:



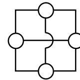
nicht möglich

Beim **L-RG** kommen nur lineare Busse vor, d.h. keine Verzweigungen. Hier sind die folgenden Verbindungen nicht (!) möglich:



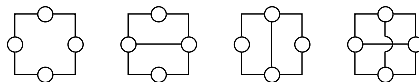
nicht möglich

Beim **non-crossover RG** ist kein Überkreuzen von Bussen erlaubt, d.h. die folgende Verbindung ist nicht (!) möglich:



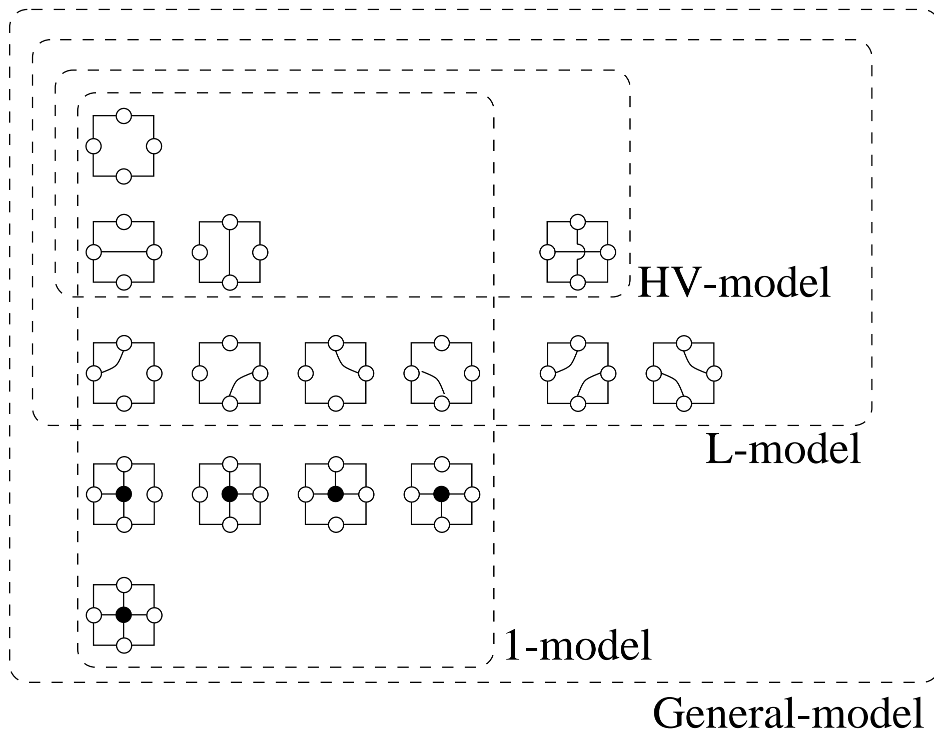
nicht möglich: cross-over

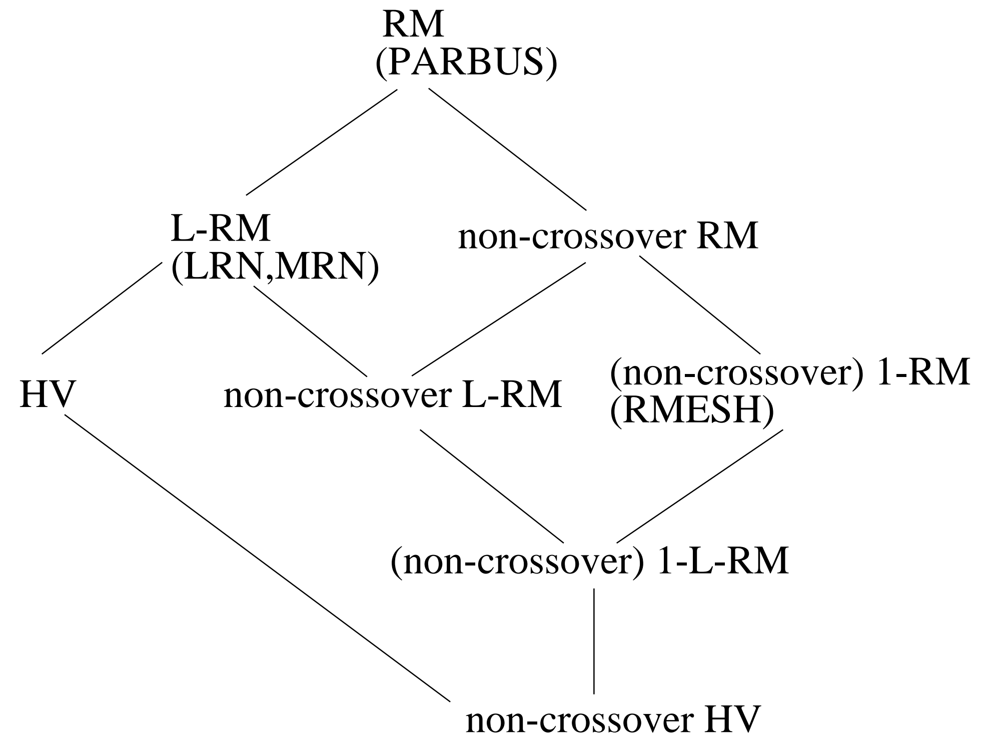
Beim **HV-RG** sind nur horizontale und vertikale Busse erlaubt, d.h. nur die folgenden Verbindungen sind möglich:



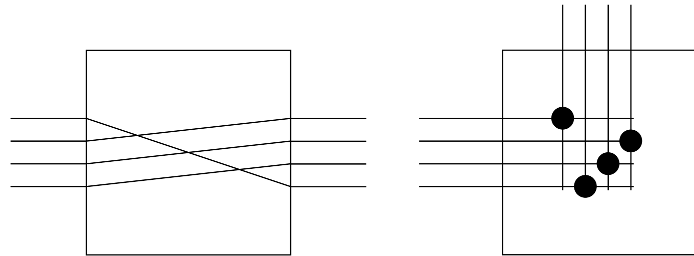
Durch Kombination der Einschränkungen ergeben sich folgende Klassen (in Klammern gebräuchliche Bezeichnungen aus der Literatur):

- RG (PARBUS)
- L-RG (LRN, MRN)
- HV-RG (HV-RN)
- non-crossover RG
- non-crossover L-RG
- non-crossover HV-RG
- (non-crossover) 1-RG (RMESH)
- (non-crossover) 1-L-RG


HV-model
L-model
1-model
General-model

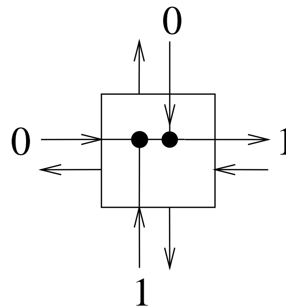

RM
(PARBUS)
L-RM
(LRN,MRN)
non-crossover RM
HV
non-crossover L-RM
(non-crossover) 1-RM
(RMESH)
(non-crossover) 1-L-RM
non-crossover HV

1. Alternative passive Schalter: Ändern der Bitreihenfolge: shift switch (cyclic), direct switch

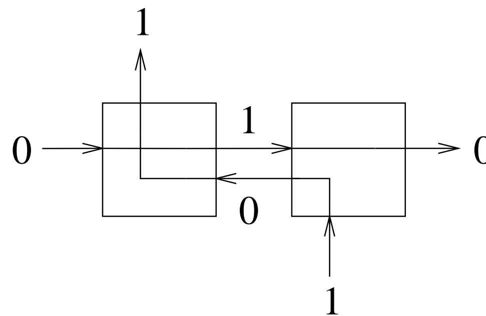


2. Aktive Schalter (für gerichtete Verbindungen):

- ODER



- NICHT



Vergleich der Berechnungsmodelle I

Simulation der PRAM auf dem RG

Theorem: Ein Algorithmus, der auf der Common CRCW PRAM mit n Prozessoren und k Speicherzellen Zeit $t(n)$ braucht, kann auf einem $k \times n$ noncrossover HV-RG in Zeit $t(n)$ simuliert werden.

Beweis: Beispielhaft gehen wir im Beweis von unserer Standardannahme ab, daß auf einem Bus auf den keiner schreibt eine Null gelesen wird.

PE $P_{1,j}$ führt dieselben arithmetischen/logischen Operationen wie Prozessor P_j der PRAM aus, $j \in [1:n]$.

PE $P_{i,1}$ speichert Inhalt der i-ten Speicherzelle M_i der PRAM, $i \in [1:k]$.

Es reicht die Schreib- und Leseoperationen zu simulieren.

Leseoperation: P_j möchte den Wert in Zelle $M_{l(j)}$ lesen, $j \in \mathcal{S} \subset [1:n]$

1. Konfiguriere Zeilenbusse
   Jedes PE $P_{i,1}$ schreibt seinen Speicherinhalt in seine Zeile.
   Alle PE's, die nicht geschrieben haben, lesen in ihrer Zeile und merken sich das Gelesene im lokalen Speicher.

2. Konfiguriere Spaltenbusse
   Jedes PE $P_{1,j}$, $j \in \mathcal{S}$, schreibt $l(j)$ in seine Spalte.
   Alle anderen PE's der ersten Zeile schreiben 0 in ihre Spalte.
   Jedes PE unterhalb der ersten Zeile liest in seiner Spalte.
   Jedes PE, das ein $l(j)$ gelesen oder gesendet hat, setzt im lokalen Speicher ein Flag $f_{i,j} = 1$.
   Desweiteren setzt jedes PE $P_{i,j}$, das ein $l(j) = i$ gelesen oder $l(j) = 1$ gesendet hat, im lokalen Speicher ein Flag $g_{i,j} = 1$.

3. Jedes PE $P_{i,j}$ mit $g_{i,j} = 1$, das nicht in der ersten Zeile ist, schreibt den Speicherinhalt der Zelle $M(j)$ in seine Spalte.
   Jedes PE der ersten Zeile mit $f_{1,j} = 1$ und $g_{1,j} = 0$ liest in seiner Spalte und speichert den gelesenen Wert im lokalen Speicher
   (Jedes PE der ersten Zeile mit $f_{1,j} = 1$ und $g_{1,j} = 1$ kennt den zu lesenden Wert bereits).

Schreiboperation: P_j möchte Wert v_j in Zelle $M_{l(j)}$ schreiben, $j \in \mathcal{S} \subset [1:n]$

1. Konfiguriere Spaltenbusse
   $P_{1,j}$ schreibt $l(j)$ auf Spaltenbus j, $j \in \mathcal{S}$.
   Alle anderen PE's der ersten Zeile schreiben 0 auf ihren Spaltenbus.
   Alle PE's, die nicht geschrieben haben, lesen in ihrer Spalte.
   Jedes PE $P_{i,j}$, das einen Wert $l(j) = i$ empfangen oder $l(j) = 1$ gesendet hat hat, setzt im lokalen Speicher ein Flag $f_{i,j}$ gleich 1.
   Alle anderen setzen ihr Flag auf 0.

2. $P_{1,j}$ schreibt v_j auf Spaltenbus j, $j \in [1:k]$.
   Jeder Prozessor $P_{i,j}$ mit $f_{i,j} = 1$, der nicht geschrieben hat, liest in seiner Spalte und merkt sich v_j im lokalen Speicher.

3. Jedes PE $P_{i,j}$ mit $f_{i,j} = 1$ konfiguriert (N,S,E,W)
   Alle anderen PE's konfigurieren (N,S,EW)
   Jedes PE $P_{i,j}$ mit $f_{i,j} = 1$, schreibt 1 in seine Zeile.
   Jedes andere PE der letzten Spalte schreibt 0 in seine Zeile.
   Jedes PE in der ersten Spalte, das nicht gesendet hat, liest in seiner Zeile.
   Jedes PE der ersten Spalte, das eine 1 empfangen oder gesendet hat, setzt im lokalen Speicher ein Flag $g_{i,1} = 1$. Alle anderen der ersten Spalte setzen $g_{i,1} = 0$.

4. Jedes PE $P_{i,j}$ mit $f_{i,j} = 1$ schreibt v_j in seine Zeile.
   Jedes PE der ersten Spalte mit $g_{i,1} = 1$, das nicht gesendet hat, liest in seiner Zeile und merkt sich das gelesene im lokalen Speicher.

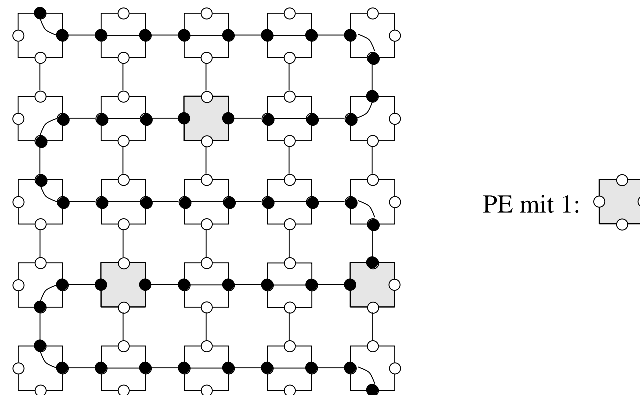
Grundlegende Algorithmen I: Logisches Oder

Gegeben: $m = n^2$ binäre Werte jeweils einer pro Prozessor im $n \times n$ RG.

Algorithmus:

1. Jedes PE mit 0 konfiguriert in Schlangenlinie.
   Jedes PE mit 1 konfiguriert (N,S,E,W).
2. Jedes PE mit 1 schreibt 1 auf den zum Anfang zeigenden Port in der Schlangenlinie.
   Falls eine 1 auf dem Anfangsport der Schlangenlinie liegt, ist das Ergebnis des logischen OR 1, anderenfalls 0.

Beispiel:



PE mit 1:

Vergleiche [Cook, Dwork,Reischuk]: Auf CREW PRAM mit beliebig vielen Prozessoren ist $\Omega(\log m)$ untere Schranke zur Berechnung des OR m binärer Werte

Ähnlich: Berechnung des logischen UND m binärer Werten in Zeit $O(1)$

XOR, Parität Gegeben: $n \times n$ RG mit einem binären Wert pro PE

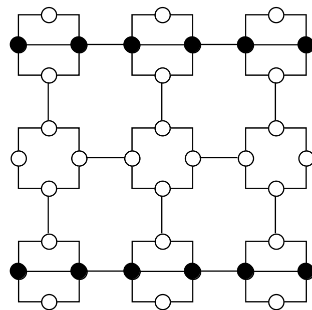
Theorem: Das XOR-Problem kann auf einem $n \times n$ L-RG in Zeit $O(1)$ gelöst werden.

Zur Vereinfachung sei n ein Vielfaches von 3. Betrachte im folgenden eine disjunkte Aufteilung des RG in 3×3 Teilgitter.

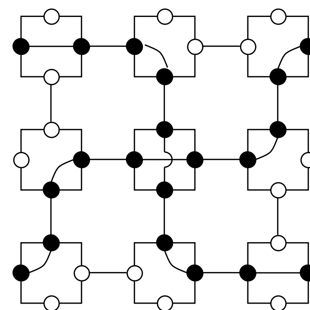
Algorithmus:

A. Löse das XOR-Problem in jedem $3 \times n$ Streifen des RG folgendermassen:

1. In jedem 3×3 Teilgitter mache folgendes
   - Berechne XOR in Zeit O(1)
   - Je nach Parität konfiguriere:



gerade Parität



ungerade Parität

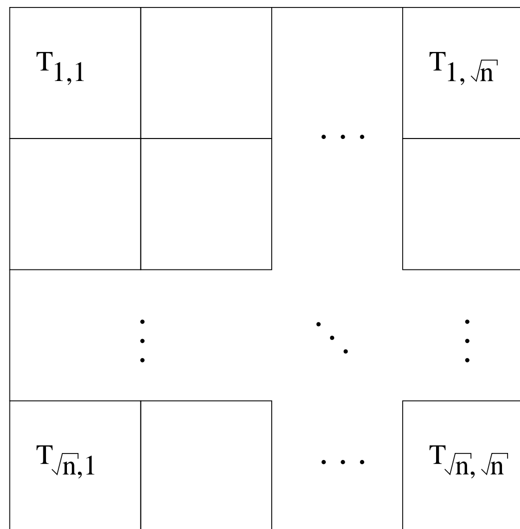
2. $P_{1,1}$ sendet 1 an Port W.
   Falls $P_{1,n}$ eine 1 auf Port E empfängt ist das Ergebnis 0, anderenfalls 1.

B. Löse das XOR Problem im letzten $n \times 3$ Streifen — analog A) — bezogen auf die Resultate aus Schritt A.

Theorem: Das XOR-Problem kann auf einem $n \times n$ non-crossover L-RG in Zeit $O(\log \log n^2)$ gelöst werden.

Algorithmus:

1. Teile das Gitter in disjunkte $\sqrt{n} \times \sqrt{n}$ Teilgitter $T_{i,j}$, $i, j \in [1 : \sqrt{n}]$ auf und löse das Problem in den Teilgittern.



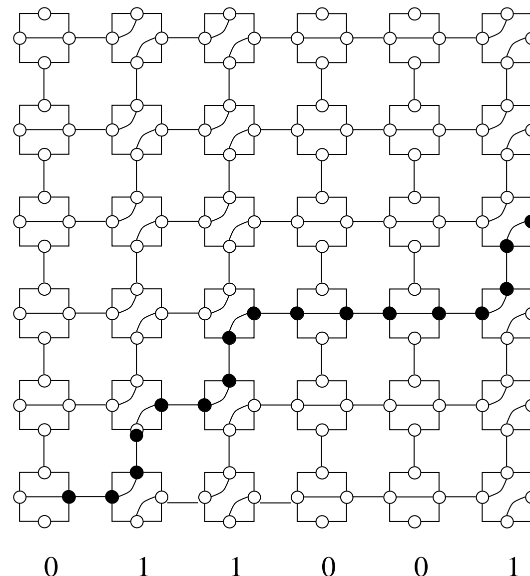
2. Sende das Result aus jedem Teilgitter $T_{i,j}$ an alle PE's in Spalte $(j-1)\sqrt{n} + i$.

3. (Annahme: Es gibt mindestens eine Spalte in der die PEs eine 0 empfangen haben. Anderenfalls benötigt man ein $n+1 \times n$ Gitter oder muß Schritt 3 entsprechend verändern).

Jeder PE, der in Schritt 2 eine 1 empfangen hat konfiguriert (NW,SE), die anderen (N,S,EW).
$P_{n,1}$ sendet eine 1 auf Port N, falls er eine 1 aus Schritt 2 hat. Anderenfalls sendet er eine 1 auf Port W.

Falls ein Prozessor mit $P_{i,n}$ mit $n-i$ gerade eine 1 empfängt, ist das Ergebnis 1; anderenfalls ist es 0.

Beispiel für Schritt 3:



Für die Laufzeit des Algorithmus erhält man ($m = n^2$):

$$T(m) = T(\sqrt{m}) + O(1)$$

und somit

$$T(m) = O(\log \log m)$$

Bem.: Schritt 3 kann auch zum Bestimmen der Anzahl der Einsen in einer Bitfolge der Länge n auf einem $n \times n$ non-crossover L-RG verwendet werden.

Korollar: Unter dem Logarithmischen Zeitmodell kann das XOR-Problem auf einem $n \times n$ non-crossover L-RG in Zeit $O(\log n^2)$ lösen.

Vergleich der Berechnungsmodelle II

Simulation des RG auf der PRAM

Theorem: [Miller et al.] Ein Algorithmus, der in Zeit $O(t(m))$ auf einem RG mit m PE's läuft, kann auf der CRCW PRAM in Zeit $O(t(m)\log m)$ simuliert werden.(Ohne Beweis)

Theorem: [Lin et al.]: Ein Algorithmus, der in Zeit $O(t(m))$ auf einem HV-RG mit m PE's läuft, kann auf der Common CRCW PRAM mit $O(m)$ zusätzlichem Speicher in Zeit $O(\alpha(m)t(m))$ simuliert werden. Dabei ist α die inverse Ackermann Funktion.

Vorbemerkungen zum Beweis:
Für eine reelle Funktion f sei die **i-te Selbstkomposition**

$$f^{(i)}(n) = f(f^{(i-1)}(n))$$

sowie

$$f^* = \min\{i \mid f^{(i)} \leq 1\}$$

Definiere Folge I_k von Funktionen:

$$I_0(n) = n - 2$$
$$I_k(n) = I^*_{k-1}$$

Somit sind z.B.

$$I_0(n) = n - 2,\ I_1(n) = \lfloor \frac{n}{2} \rfloor,\ I_2(n) = \lfloor \log n \rfloor.$$

Nun ist $\alpha(n) := \min\{i \mid I_i(n) \leq i\}$.

Die inverse Ackermann Funktion wächst extrem langsam (für alle praktisch vorkommenden Zahlen ist ihr Wert ≤ 4).

Im Beweis taucht das **Nächste-Eins Problem** auf:

Gegeben: Binärer Array $B[1 \ldots k]$.
Bestimme für jedes $i \in [1 : n]$ das größte $j < i$ mit $B[j] = 1$.

Theorem: [Berkmann, Vishkin]: Das Nächste-Eins Problem kann auf einer Common CRCW PRAM mit $\frac{k}{\alpha(k)}$ Prozessoren in Zeit $\alpha(k)$ gelöst werden. (Ohne Beweis)

Beweis (Satz von Lin et al.):

Das Gitter habe die Größe $k \times n$, $k \leq n$.

Processor P_i der PRAM simuliert PE P_{r_i,c_i} mit

$$r_i = \lceil \frac{i}{k} \rceil \qquad c_i = (i-1) \bmod n + 1$$

d.h. wenn P_{r_i,c_i} eine arithmetische/logische Operation durchführt, führt P_i die gleiche Operation durch.

Es bleibt die Lese-/Schreiboperationen auf Bussen zu simulieren.

Betrachte im folgenden Busse innerhalb einer Zeile (Für Busse innerhalb einer Spalte gehe analog vor) :

Der zusätzliche Speicher sei ein Array $B[1 \ldots k, 1 \ldots n]$.

Falls P_{r_i,c_i} in einem Takt die Ports E und W verbunden hat, schreibt P_i eine 0 in $B[r_i, c_i]$, anderenfalls eine 1.

Man erhält eine $0/1$ Folge in jeder Zeile von B, wobei zusamenhängende Nullen einen Bus darstellen. Die Arrayzelle mit einer 1, die links neben einem "Bus" aus Nullen steht, ist die Anfangszelle des Busses (Beachte: 0 am Anfang der Zeile erfordert Sonderbehandlung).

Löse das Nächste-Eins Problem in Zeit $\alpha(k)$ auf den Zeilen von B. Resultat: Jeder Prozessor kennt den Anfang des Zeilenbusses an dem er liegt (dies können auch zwei Busse sein).

Ein Prozessor, der schreiben will, schreibt in die entsprechende Anfangszelle des Busses.

Anschließend lesen alle Prozessoren, von der Anfangszelle des entsprechenden Busses.

Untere Schranken für XOR

Theorem: [Beame, Hastad] Die Berechnung des XOR von m binären Werten benötigt Zeit $\Omega(\frac{\log(m)}{\log\log m})$ auf einer Common CRCW PRAM mit polynomiell vielen Prozessoren. (Ohne Beweis)

Zusammen mit dem Satz von Lin et al. ergibt sich folgende untere Schranke:

Theorem: Das XOR von m binären Werten benötigt Zeit $\Omega(\frac{log(m)}{\alpha(m)\log\log m})$ auf einem $k \times n$ HV-RG mit $k \cdot n = m$.

Zusammen mit der Tasache, dass das XOR von m binären Werten auf einem $k \times n$ L-RG mit $k \cdot n = m$ in konstanter Zeit berechnet werden kann, folgt:

Theorem: Das L-RG Modell ist algorithmisch echt stärker als das HV-RG Modell.

Grundlegende Algorithmen II: Kompaktifizieren

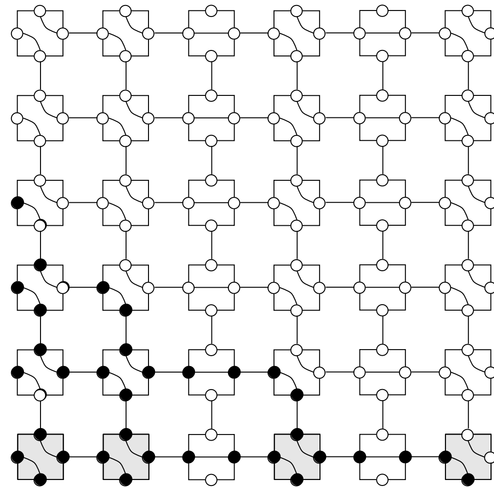
Gegeben: Folge von Zahlen $A = a_1, a_2, \ldots a_n$, wobei a_i in Prozessor $P_{n,i}$ eines $n \times n$ noncrossover L-RG steht

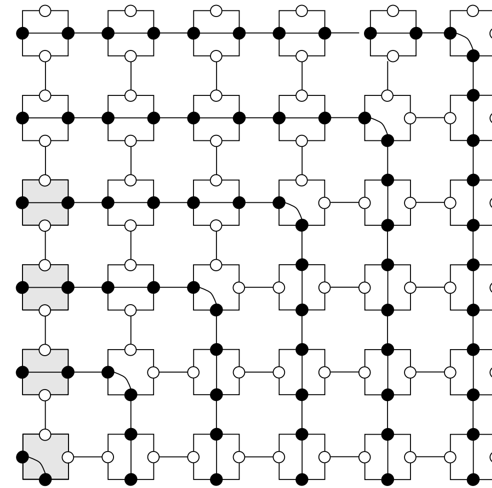
Sei b_j das $j$te a_i welches ungleich Null ist, $j \in [1 : n']$.

Problem: Kompaktifiziere Folge A, so dass b_j in PE $P_{n,j}$, $j \in [1 : n']$ steht

Algorithmus:

1. PE $P_{n,i}$ verschickt a_i in der $j$ten Spalte, $i \in [1 : n]$.
2. Jedes PE mit einem Element ungleich Null konfiguriert (NE,SW). Die anderen konfigurieren (N,S,EW).
   Jedes PE in letzte Zeile mit Element ungleich Null verschickt es nach S.
   Jedes PE der ersten Spalte liest von W.
3. $P_{i,j}$ konfiguriert:
   (N,SW,E), falls $j = n - (i - 1)$
   (N,S,EW) falls $j < n - (i - 1)$
   (NS,E,W) falls $j > n - (i - 1)$
   Jedes PE in erster Spalte mit Element ungleich Null schreibt es nach W.
   Jedes PE in letzter Zeile liest von S.





Schritt2

Schritt 3

Summe einer Folge von Bits

Gegeben: Eine Folge $A = (a_1, a_2, \ldots, a_n)$ von n Bits

Gesucht: Die Summe $x = a_1 + a_2 \ldots + a_n$

Es ist einfach die Summe auf einem $n \times n$ L-RG in Zeit $O(1)$ zu berechnen (vgl. Schritt 3 des Algorithmus zur Berechnung des logischen ODER einer Folge von n Bits auf einem $n \times n$ RG).

Im folgenden soll ein Algorithmus vorgestellt werden, der mit einem kleineren Prozessorfeld auskommt.

Gegeben sei ein $k \times n$ L-RG, wobei a_j im PE $P_{1,j}$ steht, $j \in [1 : n]$.

Idee: Sei $z \in \mathrm{IN}$. Für alle $j \in [1:n]$ definiere b_j folgendermaßen:

Falls $a_j = 1$ und $a_1 + a_2 + \ldots + a_j \bmod z = 0$, sei $b_j = 1$, anderenfalls sei $b_j = 0$. Dann gilt:

$$x = (x \bmod z) + z \cdot (b_1 + b_2 + \ldots + b_n)$$

Berechne nun rekursiv die Summe der n binären Werte $(b_1, b_2, \ldots, b_n)$ um x zu erhalten.

Ziel: Wähle z nicht zu klein, damit die Rekursionstiefe klein wird.

Problem: Moduloberechnung bezüglich einer großen Zahl ist nicht einfach.

Ausweg: Wähle z als Produkt der ersten q Primzahlen:

$$z = p_1 \cdot p_2 \cdot \ldots \cdot p_q$$

und nutze folgende Tatsache:

Falls $x \bmod p_i = y \bmod p_i$ für alle $i \in [1:q]$, dann gilt
$x \bmod p_1 \cdot p_2 \cdot \ldots \cdot p_q = y \bmod p_1 \cdot p_2 \cdot \ldots \cdot p_q$.

Für ein kleinstes solches y gilt: $y = y \bmod p_1 \cdot p_2 \cdot \ldots \cdot p_q$.

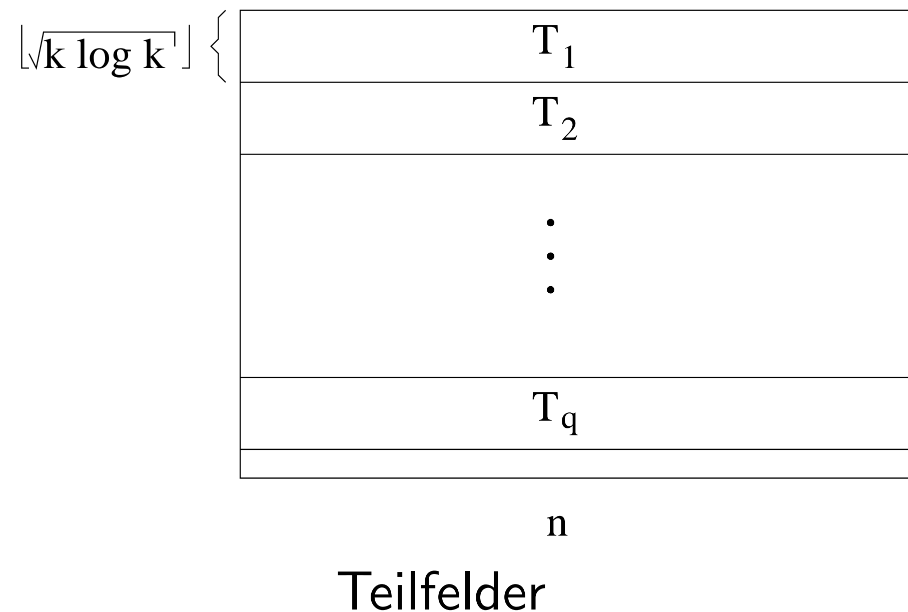
Es seien $p_1, p_2, \ldots, p_q$ die ersten q Primzahlen, so daß gilt:

$$q \leq \lfloor \sqrt{\frac{k}{\log k}} \rfloor \qquad \text{und} \qquad p_q \leq \lfloor \sqrt{k \log k} \rfloor - 1$$

Der folgende Algorithmus berechnet $x \bmod p_1 \cdot p_2 \cdot \ldots \cdot p_q$ und $p_1 \cdot p_2 \cdot \ldots \cdot p_q$ in konstanter Zeit.

Annahme: $k \geq 16$ (anderenfalls benutze $O(\log n)$ Algorithmus auf $n \times 1$ RG zur Berechnung von x).

Dazu: Teile das RG in $q \le \lfloor\sqrt{\frac{k}{\log k}}\rfloor$ viele $\lfloor\sqrt{k \log k}\rfloor \times n$ Teilgitter T_i auf.



Teilfelder

A. Berechne die ersten q Primzahlen, so daß jeder Prozessor im Teilfeld T_h die $h$te Primzahl kennt:

1. Jedes PE $P_{i,j}$ setzt Flag $f_{i,j} = 1$, falls j durch i geteilt wird.
2. $P_{1,j}$ prüft, ob j Primzahl ist, indem das ODER in der $j$ten Spalte berechnet wird, $j \le \lfloor\sqrt{k \log k}\rfloor - 1$.
3. Verschicke die Primzahlen so, daß p_j den PE's im Teilfeld T_j bekannt ist (Hinweis: Technik ähnlich wie beim Kompaktifizieren).

B. Berechne $x \bmod p_1 \cdot p_2 \cdot \ldots \cdot p_q$

1. Berechne für jede Präfixsumme von A den Wert modulo p_i im Teilfeld T_i, $i \in [1:q]$. ($\longrightarrow$ Übung)
   Das Ergebnis von $x \bmod p_i$ wird allen PE's des $i$ten Teilfeldes bekannt gemacht.

2. Berechne für die Präfixsummen der Folge $(0, 1, 1, \ldots, 1)$ der Wert modulo p_i im Teilfeld T_i, $i \in [1:q]$.
   Spalte j im Teilfeld T_i kennt jetzt den Wert $j - 1 \bmod p_i$, $j \in [1:n]$.

3. Prüfe in Spalte j von Teilfeld T_i, ob $x \bmod p_i = j - 1 \bmod p_i$ gilt, $i \in [1:q]$, $j \in [1:n]$.

4. Berechne in Spalte j, ob $x \bmod p_i = j - 1 \bmod p_i$ für alle $i \in [1:q]$ gilt, $j \in [1:n]$.

5. Berechne
   $r = \min\{j \in [1:n] \mid x \bmod p_i = j - 1 \bmod p_i, \forall i \in [1:q]\}$
   Verschicke r an alle PE's.

Es gilt: $r = x \bmod p_1 \cdot p_2 \cdot \ldots \cdot p_q$.

C. Berechne $p_1 \cdot p_2 \cdot \ldots \cdot p_q$

1. Prüfe in Spalte j, ob $j - 1 \bmod p_i = 0$ gilt für alle $i \in [1 : q]$, $j \in [1 : n]$.

2. Berechne
   $s = \min\{j \in [1 : n] \mid j \bmod p_i = 0, \forall i \in [1 : q]\}$
   Verschicke s an alle PE's.

Es gilt: $s = p_1 \cdot p_2 \cdot \ldots \cdot p_q$.

Wir benötigen folgendes Lemma

Lemma: Für k, q, p_q (wie oben gewählt) existieren $c \geq 0$ und k_0, so daß für alle $k > k_0$ gilt:

$$p_1 \cdot p_2 \cdot \ldots \cdot p_q \geq c^{\sqrt{k \log k}}.$$

Beweis: Nach dem Primzahlsatz gilt:

$$\lim_{k \to \infty} \frac{q \ln p_q}{p_q} = 1.$$

Also existiert $c \geq 0$, so daß für alle $k \geq k_0$ gilt

$$\frac{q}{2} \geq c \cdot \sqrt{\frac{k}{\log k}} \text{ und } p_{\frac{q}{2}} \geq c \cdot \sqrt{k \log k}$$

Es folgt

$$\log(p_1 \cdot p_2 \cdot \ldots \cdot p_q) \geq \log p_{\frac{q}{2}} + \log p_{\frac{q}{2}+1} + \cdots + \log p_q$$

$$\geq c \cdot \sqrt{\frac{k}{\log k}} \times (\log c + \frac{1}{2} \log k + \log \log k^{\frac{1}{2}})$$

$$\geq (\frac{c}{2}) \cdot \sqrt{k \log k}$$

Mit Hilfe des obigen Algorithmus berechne x nach folgender Rekursion:

Für alle $j \in [1:n]$ berechne b_j folgendermaßen: Falls $a_j = 1$ und

$$a_1 + a_2 + \ldots + a_j \bmod p_i = 0 \text{ für alle } i \in [1:q],$$

sei $b_j = 1$, anderenfalls sei $b_j = 0$.

Dann gilt:

$$x = (x \bmod p_1 \cdot p_2 \cdot \ldots \cdot p_q) + p_1 \cdot p_2 \cdot \ldots \cdot p_q(b_1 + b_2 + \ldots + b_n)$$

Berechne nun rekursiv die Summe der n binären Werte $(b_1, b_2, \ldots, b_n)$ um x zu erhalten.

Es sei t die Tiefe der Rekursion, dann ist t höchstens gleich dem kleinsten ganzen Wert für den $(p_1 \cdot p_2 \cdot \ldots \cdot p_q)^t \geq n$ gilt.

Nach Lemma gilt $(p_1 \cdot p_2 \cdot \ldots \cdot p_q)^t \geq n$ wenn $(c^{\sqrt{k \log k}})^t \geq n$.

Durch logarithmieren erhält man

Theorem: [Nakano] Die Summe einer Folge von n Bits kann in Zeit $O(\log n / \sqrt{k \log k})$ auf einem $k \times n$ L-RG berechnet werden.

Korollar: Die Summe einer Folge von n Bits kann in Zeit $O(1)$ auf einem $\frac{\log^2 n}{\log\log n} \times n$ L-RG berechnet werden.

Für dichte Probleminstanzen läßt sich zeigen:

Theorem: [Middendorf] Die Summe von kn Bits kann in Zeit $O(\log^\star n + \log n / \sqrt{k \log k})$ auf einem $k \times n$ L-RG berechnet werden.

Dabei ist $\log^\star n$ die Anzahl der Operationen, die man anwenden muß um von n durch fortgesetztes Logarithmieren auf eine Zahl ≤ 1 zu bekommen.

Summe einer Folge von binären Zahlen

Gegeben: Binäre Zahlen $z_1, z_2, \ldots, z_n$ mit jeweils $\log n$ Bits.

Gesucht: Summe $S = z_1 + z_2 + \ldots + z_n$

Mit Hilfe des Satzes von Nakano zeigen wir:

Theorem: Die Summe $z_1+z_2+\ldots+z_n$ von binären Zahlen $z_1, z_2, \ldots, z_n$ mit jeweils $\log n$ Bits läßt sich in Zeit $O(1)$ auf einem $\frac{\log^3 n}{\log\log n} \times n$ RG berechnen.

Beweis:

Idee: Berechne zuerst die Summe $S^{(i)}$ der $i$ten Bits der Zahlen $z_1, z_2, \ldots, z_n$. Anschließend berechne

$$S = S^{(1)} \cdot 2^0 + S^{(2)} \cdot 2^1 + \ldots + S^{(\log n)} \cdot 2^{\log n - 1}$$

wobei man das $i$te S_i Bit, $i \in [1 : 2\log n - 1]$ von S folgendermaßen berechnet

$$S_i = (S_1^{(i)} + S_2^{(i-1)} + \ldots + S_{\min\{i,\log n\}}^{(\max\{1,i-\log n\})} + \ddot{U}_{i-1}) \bmod 2$$

wobei für den Übertrag gilt

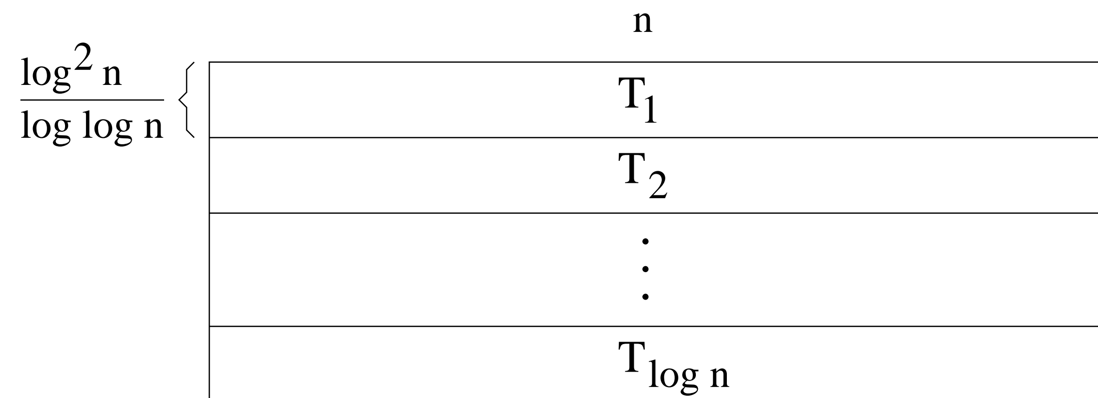
$$\ddot{U}_i = (S_1^{(i)} + S_2^{(i-1)} + \ldots + S_{\min\{i,\log n\}}^{(\max\{1,i-\log n\})} + \ddot{U}_{i-1}) DIV 2$$

und $\ddot{U}_0 = 0$.

In PE $P_{1,j}$ sei z_j gespeichert.

Algorithmus:

A. Teile das RG in $\log n$ viele Teilgitter T_i der Größe $\frac{\log^2 n}{\log\log n} \times n$ auf.

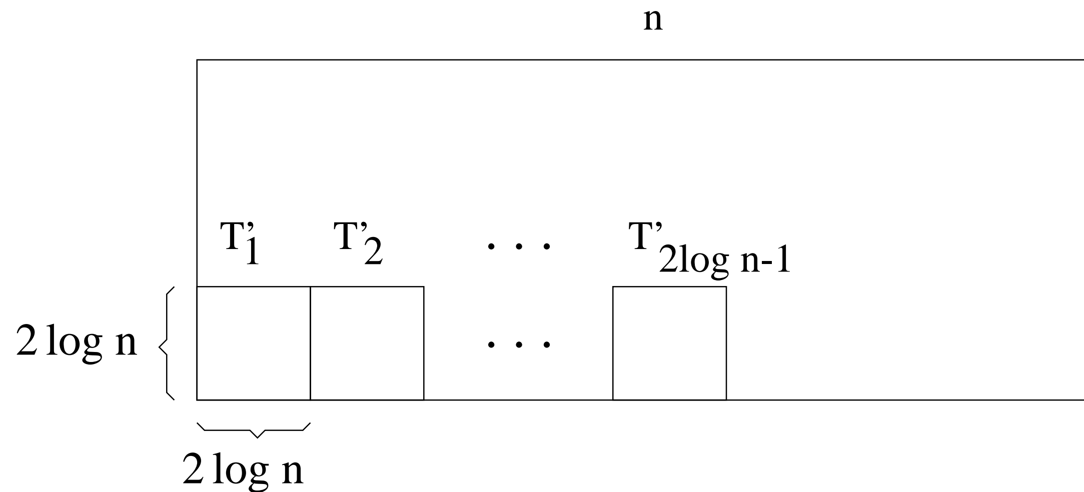


Berechne die Summen der $i$ten Bits der Zahlen $z_1, z_2, \ldots, z_n$ im Teilgitter T_i.

1. Jedes PE $P_{1,j}$ verschickt z_j in seiner Spalte.
   In jeder Spalte j merken sich die Prozessoren der oberen Zeile jedes Teilgitters T_i jeweils das $i$te Bit von z_j, $j \in [1 : n]$, $i \in [1 : \log n]$.

2. In jedem Teilgitter T_i wird in konstanter Zeit die Summe der $i$ten Bits der Zahlen $z_1, z_2, \ldots, z_n$ berechnet (vgl. Korollar zum Satz von Nakano).

B. Lege $2\log n - 1$ viele $2\log n \times 2\log n$ Teilgitter T_i' fest:



Im Teilgitter T_i' wird S_i berechnet.

Den Übertrag dabei bekommt das Teilgitter T_i' vom Teilgitter T_{i-1}' und gibt seinen Übertrag an T_{i+1}' weiter.

1. Über die Spaltenbusse erhält jedes PE in der $j$ten Spalte im Teilgitter T_i', $j \in [1 : \min\{i, \log n\}]$, $i \in [1 : 2n - 1]$ das $j$te Bit der Bitfolge

$$S_1^{(i)} + S_2^{(i-1)} + \ldots + S_{\min\{i,\log n\}}^{(\max\{1,i-\log n\})}$$

   Alle anderen PEs im Teilgitter T_i' erhalten eine Null.

2. PE's in den ersten $\log n$ Spalten jedes Teilgitters T_i' konfigurieren

   (NW,SE), falls das PE eine 1 erhalten hat
   (N,S,EW) sonst

   PE's in der $u$ten Spalte und $v$ten Zeile, $u \in [\log n + 1 : 2\log n]$, $u \in [1 : 2\log n]$ eines Teilgitters T_i' konfigurieren

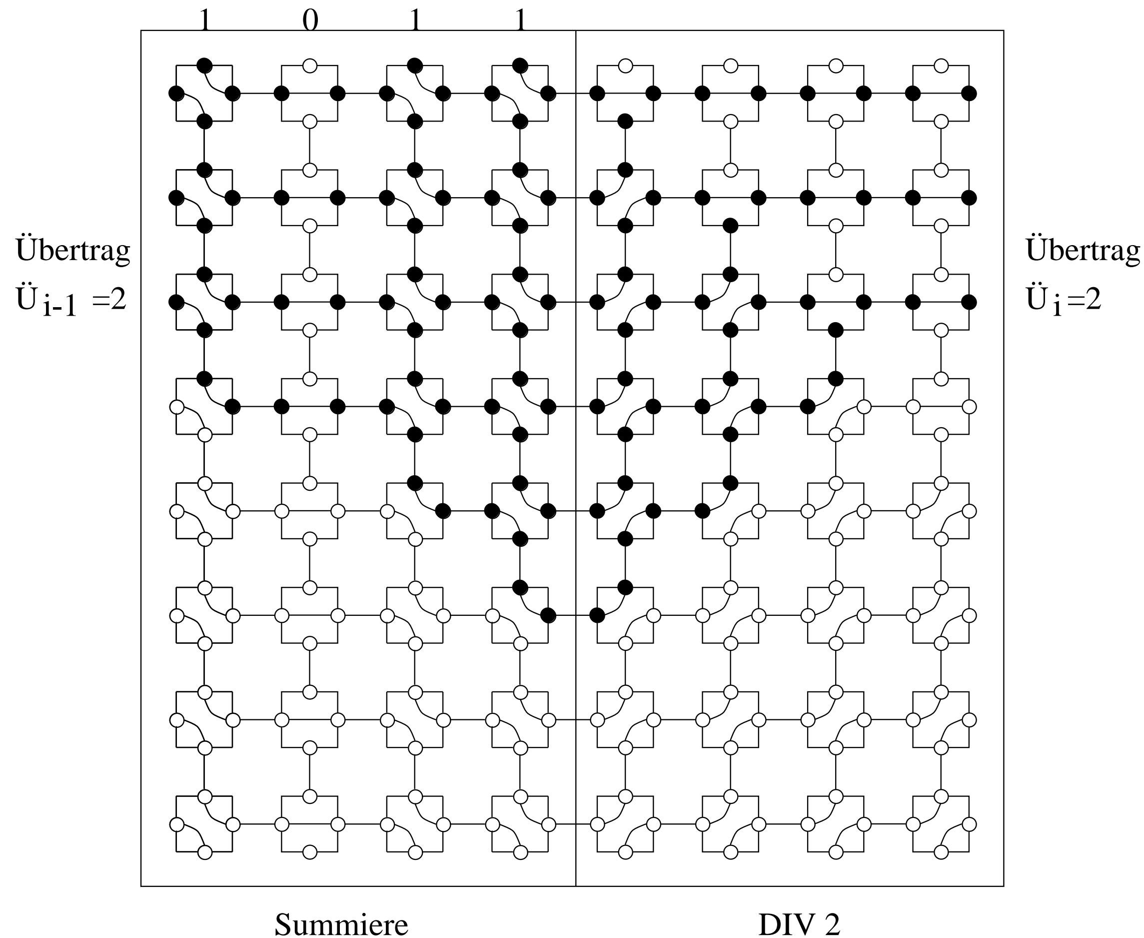
   (NE,SW), falls $u > v$
   (N,S,EW) sonst

   Ausnahme: Das PE in der ersten Zeile und ersten Spalte von T_1' konfiguriert (NSW,E), falls es eine 1 hat, anderenfalls (NW,S,E).

   PEs, die in der ersten Zeile eines Teilgitters T_i' stehen und eine 1 haben, senden eine 1 nach N.

   Ausnahme: Das PE in der ersten Zeile und ersten Spalte von T_1' sendet in jedem Fall eine 1 nach N.
   Alle PE's in der $\log n$ten Spalte eines Teilgitters T_i' lesen von E.

3. Bestimme nun in der $\log n$ten Spalte jedes Teilgitters T_i' das unterste PE, das eine 1 empfangen hat.
   Hat dieses PE einen ungeraden Zeilenindex inerhalb von T_i', so ist $S_i = 0$.
   Anderenfalls ist $S_i = 1$.



Beispiel: Berechnung in T_i'

Sortieren

Gegeben: Eine Folge von n Zahlen $z_1, z_2, \ldots, z_n$

Problem: Sortiere die Zahlen

Theorem: [Nigam, Sahni] Eine Folge von n Zahlen kann in Zeit $O(1)$ auf einem $n \times n$ 1-L-RG sortiert werden.

Strategie: Nimm einen Sortier-Algorithmus für ein "normales" $\sqrt{n} \times \sqrt{n}$ Prozessorgitter, der nur aus konstant vielen parallelen Zeilen- und Spaltensortier-Schritten besteht und simuliere jeden parallelen Zeilen- und Spaltensortier-Schritt in konstanter Zeit auf dem RG.

Wir benutzen das RotateSort Verfahren [Marberg, Gafni] für ein "normales" $N \times N$ Gitter. RotateSort im folgenden vorgestellt.

Beim Beweis der Korrektheit von Rotate-Sort verwenden wir das 0/1-Prinzip (vgl. z.B. Cormen, Leiserson, Rivest: Introduction to Algorithms).

0/1-Prinzip: Wenn ein nur auf Vegleichen basierender Algorithmus mit n Eingaben sämtliche 0/1-Folgen (der Länge n) sortiert, so sortiert er Folgen beliebiger Zahlen korrekt.

Annahme: $N = 2^{2t}$, $t \in$ IN.

Im folgenden partitionieren wir das Gitter in

$\sqrt{N}$ Teilgitter der Größe $\sqrt{N} \times N$: horizontale Teilgitter.
$\sqrt{N}$ Teilgitter der Größe $N \times \sqrt{N}$: vertikale Teilgitter.
N Teilgitter der Größe $\sqrt{N} \times \sqrt{N}$: Blöcke.

Eine Zeile (Block) in der sowohl Einsen als auch Nullen stehen heißt **gemischte** Zeile (Block).

Rotate-Sort benutzt drei Bausteine:

Balancieren, Block-Rotieren, LR-Sortieren

A. **Balancieren** auf $N_1 \times N_2$ Teilgitter

1. Sortiere jede Spalte abwärts
2. Rotiere jede Zeile i um $(i \bmod N_2)$ Stellen nach rechts
3. Sortiere jede Spalte abwärts

Es gilt: Nach Anwendung von Balancieren auf $N_1 \times N_2$ Teilgitter gibt es höchstens N_2 gemischte Zeilen.

Denn ($N_1 > N_2$, sonst trivial): Schritt 2 bewirkt, dass die Elemente jeder Spalte so auf die Spalten verteilt werden, dass die Anzahl der Nullen die jede Spalte von einer bestimmten Spalte bekommt sich höchstens um eins unterscheidet.

Also unterscheidet sich die Anzahl der Nullen zwischen zwei Spalten nach Schritt 2 um höchstens N_2.

B. **Block-Rotieren**

1. Rotiere jede Zeile i um $(i \cdot \sqrt{N} \bmod N)$ Stellen nach rechts
2. Sortiere jede Spalte abwärts

Es gilt: Nach Anwendung von Block-Rotieren auf ein $N \times N$ Gitter mit k gemischten Blöcken, gibt es höchstens k gemischte Zeilen.

Denn: In Schritt 1 bekommt jede Spalte von jedem Block genau ein Element.

C. **LR-Sortieren**

1. Sortiere jede gerade Zeile nach rechts und jede ungerade Zeile nach links
2. Sortiere jede Spalte abwärts

Es gilt: Nach Anwendung von LR-Sortieren auf ein $N \times N$ Gitter mit k gemischten Zeilen gibt es höchstens $\lceil \frac{k}{2} \rceil$ gemischte Zeilen.

Denn: Für je zwei benachbarte gemische Zeilen unterscheidet sich nach Schritt 1 die Anzahl der Nullen in den Spalten dieser zwei Zeilen um höchsten eins.

Beachte: Die gemischten Zeilen liegen nach Anwendung von Balancieren, Block-Rotieren oder LR-Sortieren jeweils benachbart (unter Zeilen mit Nullen, über Zeilen mit Einsen).

Algorithmus: RotateSort

1. Balanciere jedes vertikale Teilgitter
   (es gibt danach höchstens $2\sqrt{N}$ gemischte Blöcke)
2. Block-Rotiere das Gitter
   (es gibt danach höchstens $2\sqrt{N}$ gemischte Zeilen)
3. Balanciere jedes horizontale Teilgitter (wie gekipptes $N \times \sqrt{N}$ Gitter)
   (es gibt danach höchstens 6 gemischte Blöcke)
4. Block-Rotiere das Gitter
   (es gibt danach höchstens 6 gemischte Zeilen)
5. LR-Sortiere das Gitter dreimal
   (es gibt danach höchstens eine gemischte Zeile)
6. Sortiere alle Zeilen (lexikographisch oder in Schlangenlinie)

Gegeben sei jetzt ein $n \times n$ RG, wobei die Zahl z_i jeweils im PE $P_{1,i}$, $\in [1:n]$.

Um RotateSort für ein $N \times N$ Gitter auf einem $n \times n$ RG mit $n = N^2$ durchführen zu können, simulieren wir das Sortieren/Rotieren einer Zeile/Spalte des Gitters in Zeit $O(1)$ auf einem $N \times n$ (bzw. $n \times N$) Teilgitter des RG.

Im folgenden betrachte nur das Sortieren der Spalten (Da das Sortieren der Zeilen und Rotieren ähnlich geht) .

Annahme: $z_1, z_2, \ldots, z_n$ seien die Zahlen auf dem Gitter in row-major Reihenfolge.

Die Zahlen in den Spalten/Zeilen des Gitters bezeichnen wir als Pseudodpalten/Pseudozeilen.

Algorithmus: Spalten sortieren

A. Betrachte eine Partition des RG's in N viele $n \times N$ Teilgitter T_i.

1. PE $P_{1,i}$ verschickt z_i entlang seiner Spalte, $\in [1:n]$
2. $P_{i,i}$ verschickt z_i entlang seiner Zeile, $\in [1:n]$
   Jedes PE $P_{i,j}$ kennt nun die Zahl z_j, $i, j \in [1:n]$.
3. In der $j$ten Spalte jedes Teilgitters T_i wird nun die $j$te Zahl der $i$ten Pseudospalte ($z_{(i-1)N+j}$) in der ganzen Spalte verschickt.
   In jeder Zeile des Teilgitters T_i befindet sich nun die $i$te Pseudospalte.

B. Betrachte nun eine Partition jedes Teilgitters T_i in $N \times N$ Teilgitter $T'_{i,j}$. In $T'_{i,j}$ wird der Rang des $j$ten Elementes $z_{(i-1)N+j}$ innerhalb der $i$ten Pseudospalte ermittelt.

Skizze: (Genaues $\Longrightarrow$ Übung)

1. Vergleiche im $j$ten PE der ersten Zeile von $T'_{i,j}$ die Zahl $z_{(i-1)N+j}$ mit jedem anderen Element $z_{(i-1)N+h}$ der Pseudospalte $h \in [1:N]$.
2. Setze ein Flag falls , $z_{(i-1)N+h} < z_{(i-1)N+j}$ oder $z_{(i-1)N+h} = z_{(i-1)N+j}$ und $h < j$.
3. Summiere über die Folge der Flags.

C. Schicke von jedem Teilgitter $T_{i,j}$ die Zahl $z_{(i-1)N+j}$ an den entsprechenden Empfänger in der ersten Zeiles des RG.

Korollar: Eine Folge Z von $k \cdot n$ Zahlen kann in Zeit $O(k \log k)$ auf einem $n \times n$ 1-L-RG sortiert werden.

Beweis:

Die kn Zahlen seien in den PE's der ersten k Zeilen des RG gespeichert.

Führe $\log k$ mal die folgenden Schritte durch:

1. Sortiere die Zeilen abwechselnd nach links und rechts.

Zeit: $O(k)$ ($O(1)$ für jeder Zeile nach Satz von Nigam und Sahni)

2. Sortiere die jeweils k Elemente in jeder Spalte (k OETS-Schritte)

Zeit: $O(k)$