Algorithmen 2

Kapitel: Parallele Algorithmen

Thomas Worsch

Fakultät für Informatik
Karlsruher Institut für Technologie

Wintersemester 2017/2018

Überblick

Warum Parallelverarbeitung

Warum Parallelverarbeitung

Zeitersparnis: p Computer, die gemeinsam an einem Problem arbeiten, lösen es bis zu p mal so schnell.

- *aber* gute Koordinationsalgorithmen nötig

Warum Parallelverarbeitung

Zeitersparnis: p Computer, die gemeinsam an einem Problem arbeiten, lösen es bis zu p mal so schnell.

- *aber* gute Koordinationsalgorithmen nötig

Kommunikationsersparnis: wenn Daten verteilt anfallen, kann man sie auch verteilt (vor)verarbeiten

Energieersparnis: zwei Prozessoren mit halber Taktfrequenz brauchen weniger als eine voll getakteter Prozessor.

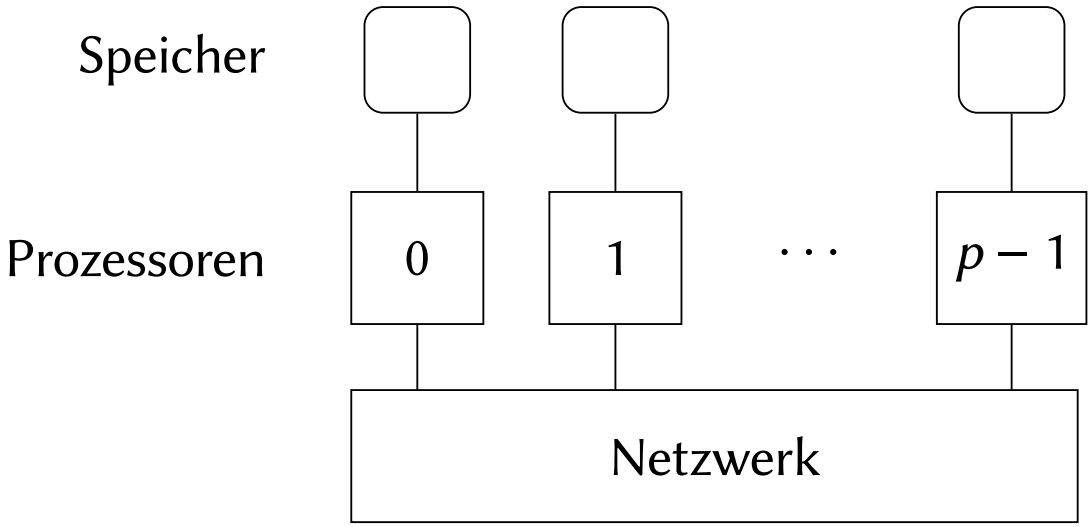
Speicherbeschränkung mehr Prozessoren haben mehr Hauptspeicher, mehr Cache, …

Verschiedene Modelle für Parallelverarbeitung

- es gibt *viiiiiiiiele*
  - konzeptionell sehr unterschiedlich (auf den ersten Blick)
  - Parallelverarbeitung zum Teil «versteckt»
- in dieser Vorlesung zwei Standardmodelle
  - Prozessornetzwerk mit *Nachrichtengekopplung*
  - *parallele Registermaschinen* mit Speicherkopplung

Nachrichtengekoppelter Parallelrechner

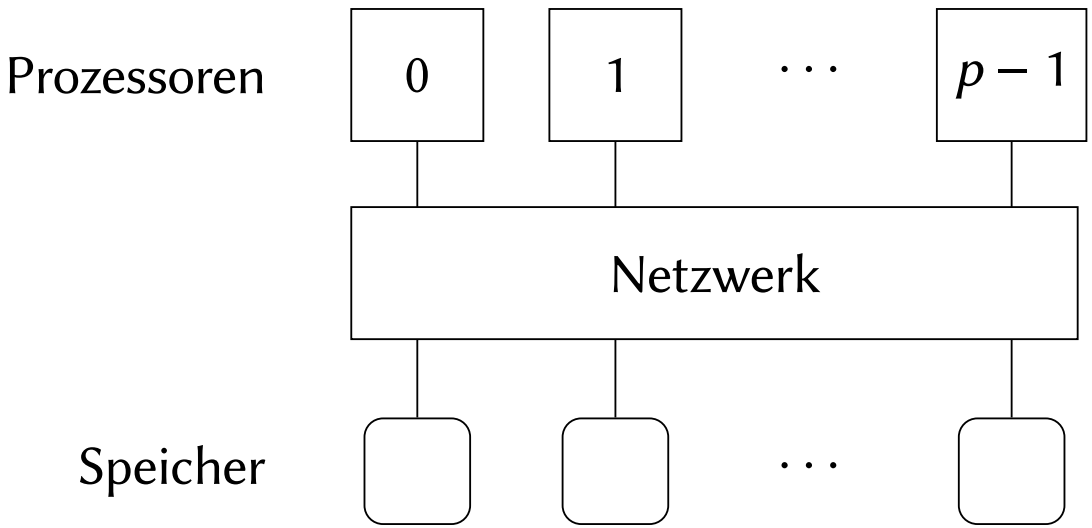
- Prozessoren: *random access machines*
  - «normale CPUs» mit lokalem Speicher
  - jedes Speichermodul nur von «seinem» Prozessor zugreifbar
  - nennen wir auch *PE (processing element)*
- *Datenaustausch*
  - über Netzwerk
  - *Nachrichten* von einem Prozessor zu einem anderen



Parallelrechner mit globalem Speicher

ein möglicher Aufbau

- Prozessoren wie üblich
  - aber kein «lokaler» Speicher
  - jedes Speichermodul von «allen» Prozessoren zugreifbar
- *Datentransport*
  - über Netzwerk
  - zwischen einer CPU und einem Speicher



Nachrichtenkopplung versus Speicherkopplung

potenzielle Nachteile von Nachrichtenkopplung

- für Datentransport Beteilgung von *zwei* Prozessoren
  - für Empfänger mitunter «unpassend»
- Parallelismus muss explizit programmiert werden

potenzielle Nachteile von Speicherkopplung

- *Skalierbarkeit:* große Prozessorzahlen sinnvoll?
- Kostenmaß bei *Speicherzugriffskonflikten?*

daher oft gute Strategie:

- Entwurf für verteilten Speicher
  - deckt viel breiteren Bereich ab
- Implementierung u. U. für gemeinsamen Speicher

Parallele Beispielalgorithmen

einfache, aber wichtige, algorithmische Probleme

- Datenumverteilung
- Präfixsummen
- Sortieren
- …

Überblick

Überblick

Modell für Nachrichtenaustausch

ähnlich wie bei MPI

- Netzwerk
  - Punkt-zu-Punkt
  - vollständig verknüpft
  - Nachrichten überholen sich nicht
  - voll-duplex
- jeder Prozessor kann maximal gleichzeitig
  - eine Nachricht an einen beliebigen Empfänger senden
    - SEND(*smsg*, *to*)
  - eine Nachricht von einem beliebigen Sender empfangen
    - *rmsg* ← RECV(*from*)
  - oder beides gleichzeitig
    - *rmsg* ← SENDRECV(*smsg*, *to*, *from*)
- wer MPI kennt, möchte mehr spezifiziert haben …

Kostenmodell für Nachrichtenaustausch

Dauer für Senden oder Empfangen von ℓ Bytes

$$T_{comm}(\ell) = T_{start} + \ell \cdot T_{byte}$$

- in der Praxis meist $T_{byte} \ll T_{start}$
- ignoriert z. B. «Abstand» zwischen Sender und Empfänger
  - auf gleicher CPU?
  - auf gleichem Mainboard?
  - in gleichem Rack?

Programmiermodell

- *SPMD:* single program multiple data
  - alle PEs führen das gleiche Programm aus
  - Unterscheidung durch «Ränge»
    - paarweise verschiedene PE-Nummern
- Beispiel:

```
parHelloWorld()
1  p = commSize()
2  iPE = commRank()
3  print("I am PE", iPE, "of", p, "PEs")
```

mögliche Ausgabe

"*I am PE* 1 *of* 3 *PEs*"
"*I am PE* 2 *of* 3 *PEs*"
"*I am PE* 0 *of* 3 *PEs*"

Überblick

Reduktion

- $\otimes$ sei binäre assoziative Operation auf Menge M
  - nicht notwendig kommutativ
  - Beispiele $x \otimes y = x + y$, $x \otimes y = \min(x, y)$, $x \otimes y = x$, $x \otimes y = y$
- für $x = (x_0, \ldots, x_{p-1}) \in M$ definiere $R_\otimes(x) = \bigotimes_{i<p} x_i = x_0 \otimes \cdots \otimes x_{p-1}$

Parallele Berechnung für Reduktion

Satz

Wenn $\otimes$ eine binäre assoziative Operation ist und
p Elemente $x_0, \ldots, x_{p-1}$ auf p PEs verteilt sind,
dann kann man $\bigotimes_{i<p} x_i$ in Zeit $\Theta(\log p)$ auf PE 0 berechnen.

Parallele Reduktion: Algorithmusidee

- für $0 \leq i < p$ liege x_i auf PE i
- Idee für den Fall $p = 2^k$:

$$\bigotimes_{i<p} x_i = \bigotimes_{i<p/2} x_i \otimes \bigotimes_{i<p/2} x_{i+p/2}$$

  und Rekursion
- (Bild live)

Parallele Reduktion: Algorithmus

```
1   active ← 1
2   sum ← x_i
3   for k ← 0 to ⌊log p⌋
4       if active
5           if bit k of i
6               SEND(sum, i − 2^k)  // bit k is 0
7               active ← 0
8           else if i + 2^k < p
9               sum′ ← RECV(i + 2^k)
10              sum ← sum ⊗ sum′
11  // result is in sum of PE 0
```

zu jedem SEND ein RECV und umgekehrt

Parallele Reduktion: Algorithmus (2)

Bild live

Parallele Reduktion: Analyse

Reduktion von n Elementen

- sequenziell: Zeit $\Theta(n)$ (optimal)
- auf $p = n$ PEs
  - Laufzeit $\Theta(\log n)$
  - Beschleunigung $\Theta(\frac{n}{\log n})$
    - «ideal» wäre Beschleunigung $\Theta(p) = \Theta(n)$
- wie kann man Beschleunigung noch verbessern?

Parallele Reduktion mit $p < n$

- jedes PE i hat n/p Datenelemente
- berechnet zuerst $s_i = \bigotimes$ der lokalen Elemente
- anschließend parallele Reduktion der s_i

- Laufzeit $\Theta(n/p) + \Theta(\log p)$
- Beschleunigung

$$\frac{\Theta(n)}{\Theta(n/p + \log p)}$$

- für $p \in \Theta(n/\log n)$ ist die Beschleunigung $\Theta(p)$

Anmerkungen

- Bäume sind des öfteren nützlich
- *Brent's Prinzip:* «Durch Verringerung der Prozessorzahl p kann man die Beschleunigung in die Nähe von p bringen.»
  - gleich noch etwas präziser

Überblick

Analyse paralleler Programme (1)

was interessiert?

- Laufzeit
- Beschleunigung
- verrichtete Arbeit

Analyse paralleler Programme (1)

was interessiert?

- Laufzeit
- Beschleunigung
- verrichtete Arbeit

- $T_{par}(I, p)$ Laufzeit Probleminstanz I bearbeitet mit p PEs
- $T_{seq}(I)$ Laufzeit Probleminstanzen Größe n mit «bestem (bekannten) Algorithmus»
- «im allgemeinen» $T_{seq}(I) < T_{par}(I, 1)$
  - aber par. Alg. kann leicht (?) modifiziert werden:
    if $p = 1$ **then** $A_{seq}(I)$ **else** $A_{par}(I)$ **fi**

Analyse paralleler Programme (2)

- *Beschleunigung* (*speedup*)

$$S(I,p) = \frac{T_{seq}(I)}{T_{par}(I,p)}$$

- Beschleunigung vergröbert

$$S(n,p) = \inf\{S(I,p) \mid n = |I|\}$$

- bequemer Spezialfall: für Instanzen I, I' gleicher Größe n ist $T_{par}(I,p) = T_{par}(I',p) = T_{par}(n,p)$ und $T_{seq}(I) = T_{seq}(I') = T_{seq}(n)$
- dann $S(n,p) = T_{seq}(n)/T_{par}(n,p)$
  - *Im allgemeinen gilt das nicht!* (Übung)

Analyse paralleler Programme (3)

- Simulation von p Prozessoren durch 1 Prozessor:
  - reihum immer «ein Stückchen» von Prozessor i
- «möglichst guter» seq. Alg. nie langsamer als $O(p \cdot T_{par}(I, p))$
- also stets $\frac{T_{seq}(I)}{T_{par}(I,p)} \in O(p)$ und $S(n, p) \in O(p)$

Analyse paralleler Programme (4)

- *Effizienz:*

$$E(n,p) = \frac{S(n,p)}{p}$$

- wegen $S(n,p) \in O(p)$ stets $E(n,p) \in O(1)$
- kurios, aber möglich: $E(p) > 1$ «superlinearer Speedup»
  - wie kann das sein?
- Ziel: $E(p) \in \Theta(1)$
-

$$E(n,p) = \frac{T_{par}(n,p)}{p \cdot T_{seq}(n)}$$

Analyse paralleler Programme (5)

- *Arbeit:*

$$W(n,p) = p \cdot T_{par}(n,p)$$

- dann

$$E(n,p) = \frac{S(n,p)}{p} = \frac{T_{seq}(n)}{T_{par}(n,p) \cdot p} = \frac{T_{seq}(n)}{W(n,p)}$$

Analyse der parallelen Reduktion mit $p < n$

jetzt präziser formuliert

- sequenzielle Reduktion von n/p Elementen
- anschließend parallele Reduktion der p partiellen Summen

- Laufzeit $T_{par}(n,p) = \Theta(n/p) + \Theta(\log p)$
- Beschleunigung $S(n,p) = \frac{\Theta(n)}{\Theta(n/p+\log p)}$
- Effizienz $E(n,p) = \frac{\Theta(n)}{\Theta(n+p\log p)}$
- für $p = n$:
  $S(n,n) \in \Theta(\frac{n}{\log n})$ und $E(n,n) \in \Theta(\frac{1}{\log n})$
- für $p \in \Theta(n/\log n)$:
  $S(n,p) = \Theta(p)$ und $E(n,p) \in \Theta(1)$
- größere Effizienz für kleinere p Brents Prinzip

Überblick

Präfixsummen

- $\otimes$ sei binäre assoziative Operation auf Menge M
  - nicht notwendig kommutativ
  - evtl. neutrales Element e
- für $x = (x_0, \ldots, x_{p-1}) \in M$ definiere
  $P_\otimes(x) = y = (y_0, \ldots, y_{p-1})$ wobei für $0 \leq i < p$:
  - $y_i = \bigotimes_{k \leq i} x_k$ oder
  - $y_i = \bigotimes_{k < i} x_k$ (falls neutrales Element e; $\bigotimes_{k<0} x_i = e$)
- also
  - $y_0 = x_0$ und $y_{i+1} = y_i \otimes x_{i+1}$,
    also
  - $y_1 = x_0 \otimes x_1$
  - $y_2 = x_0 \otimes x_1 \otimes x_2$
  - $y_3 = x_0 \otimes x_1 \otimes x_2 \otimes x_3$
  - …

Präfixsummen: Hyperwürfel-Algorithmus

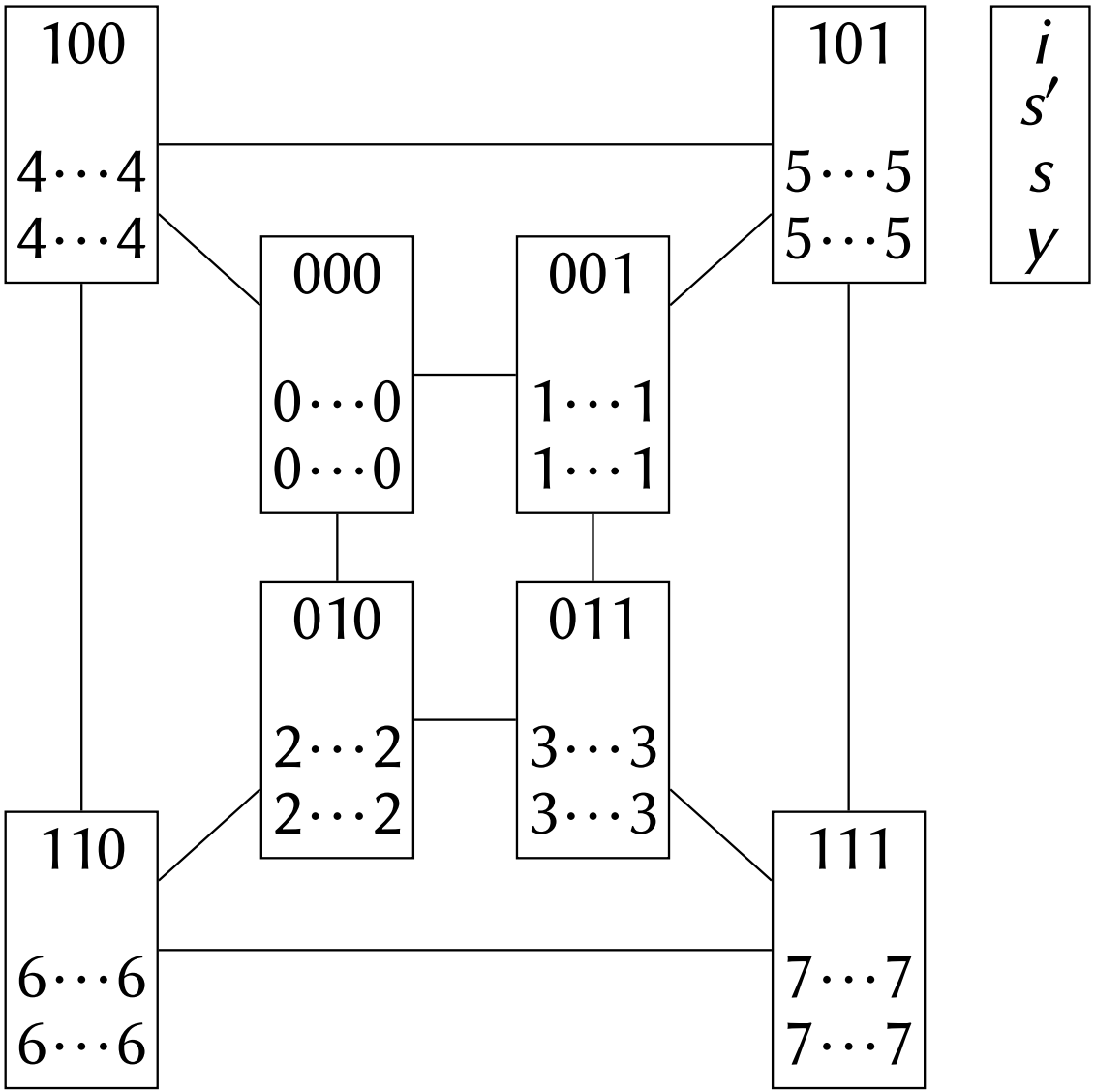
- um es bequem zu haben:
  - $p = n = 2^d$
  - kommutatives $\otimes$
- PE-«Koordinaten» $0 \leq i < 2^d$
  - repräsentiert als Bitvektoren $i = (i_{d-1} \cdots i_0)$ mit $i_j \in \{0, 1\}$
    - i_k: das k-te Bit von rechts in i
  - Kommunikation $i \leftrightarrow i'$ nur bei Hammingdistanz 1, also $i' = i \oplus 2^k \pmod{2^d}$
    - $\rightsquigarrow$ Hyperwürfel

Präfixsummen: Hyperwürfel-Algorithmus

für den Fall $p = n = 2^d$ und kommutatives $\otimes$

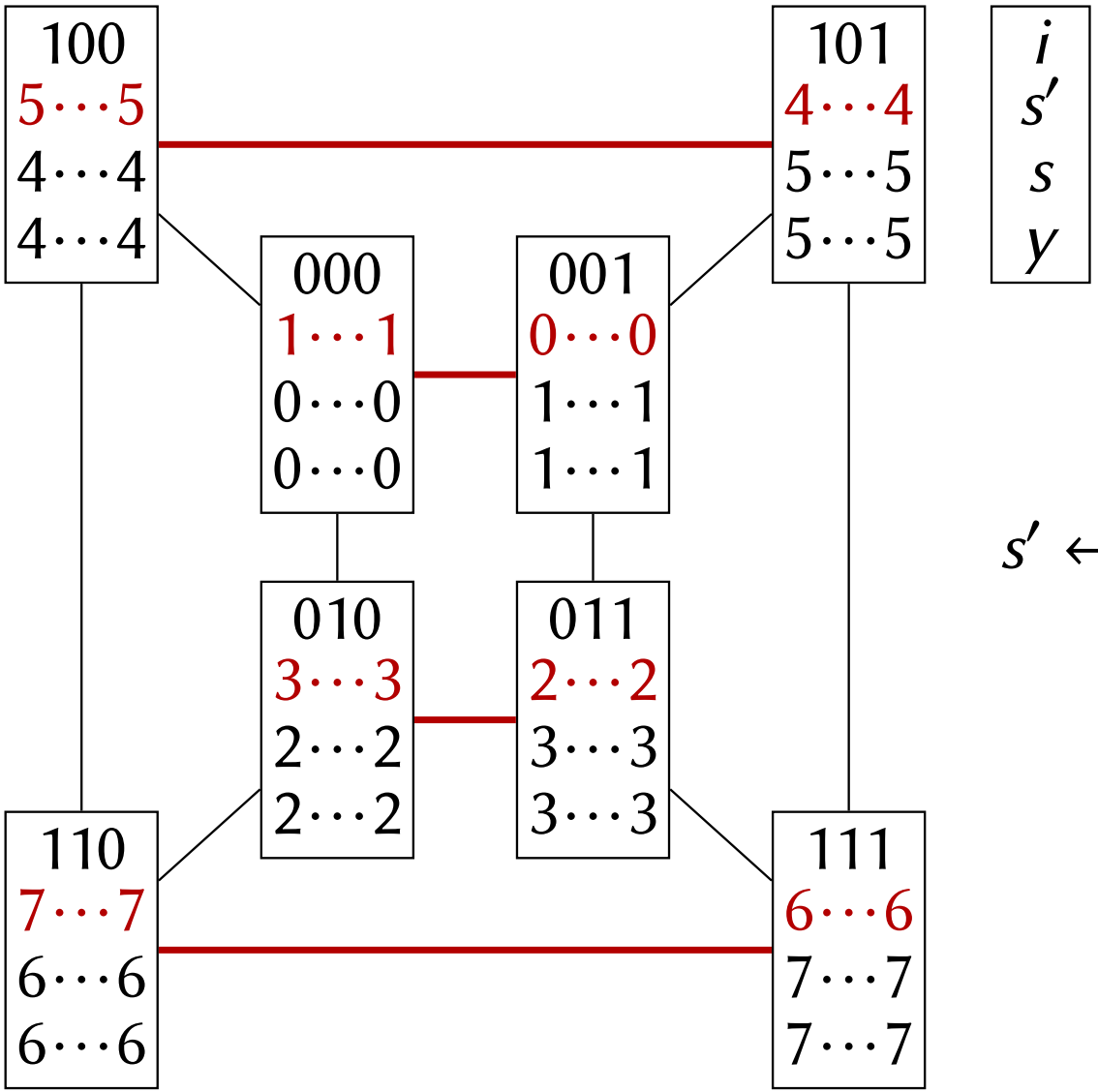
```
PREFIXSUM(x, ⊗)
1  y ← x  // auf PE i liegt x_i
2  s ← x  // für Summe von Elementen in Unterwürfel
3  for k ← 0 to d − 1
4        s' ← SENDRECV(s, i ⊕ 2^k, i ⊕ 2^k)
5        s ← s ⊗ s'
6        if i_k = 1
7              y ← y ⊗ s'
8  // auf PE i liegt y_i = x_0 ⊗ ··· ⊗ x_i
```

Präfixsummen: Hyperwürfel-Algorithmus (Init)



$$k \cdots m = \bigotimes_{j=k}^{m} x_j$$

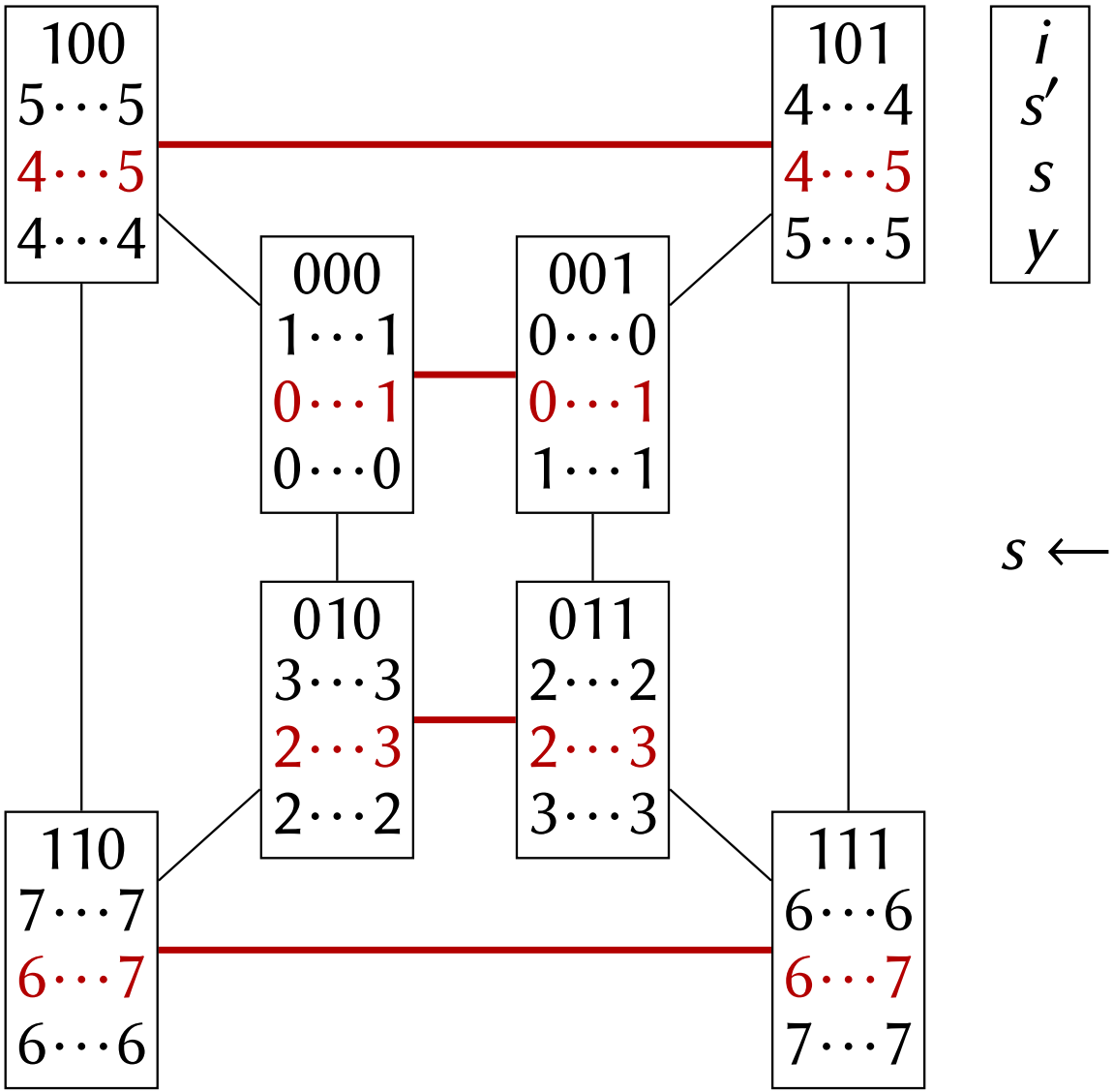
Präfixsummen: Hyperwürfel-Algorithmus ($k = 0$)



$s' \leftarrow \text{SENDRECV}(s, i \oplus 2^0, i \oplus 2^0)$

$k \cdots m = \bigotimes_{j=k}^{m} x_j$

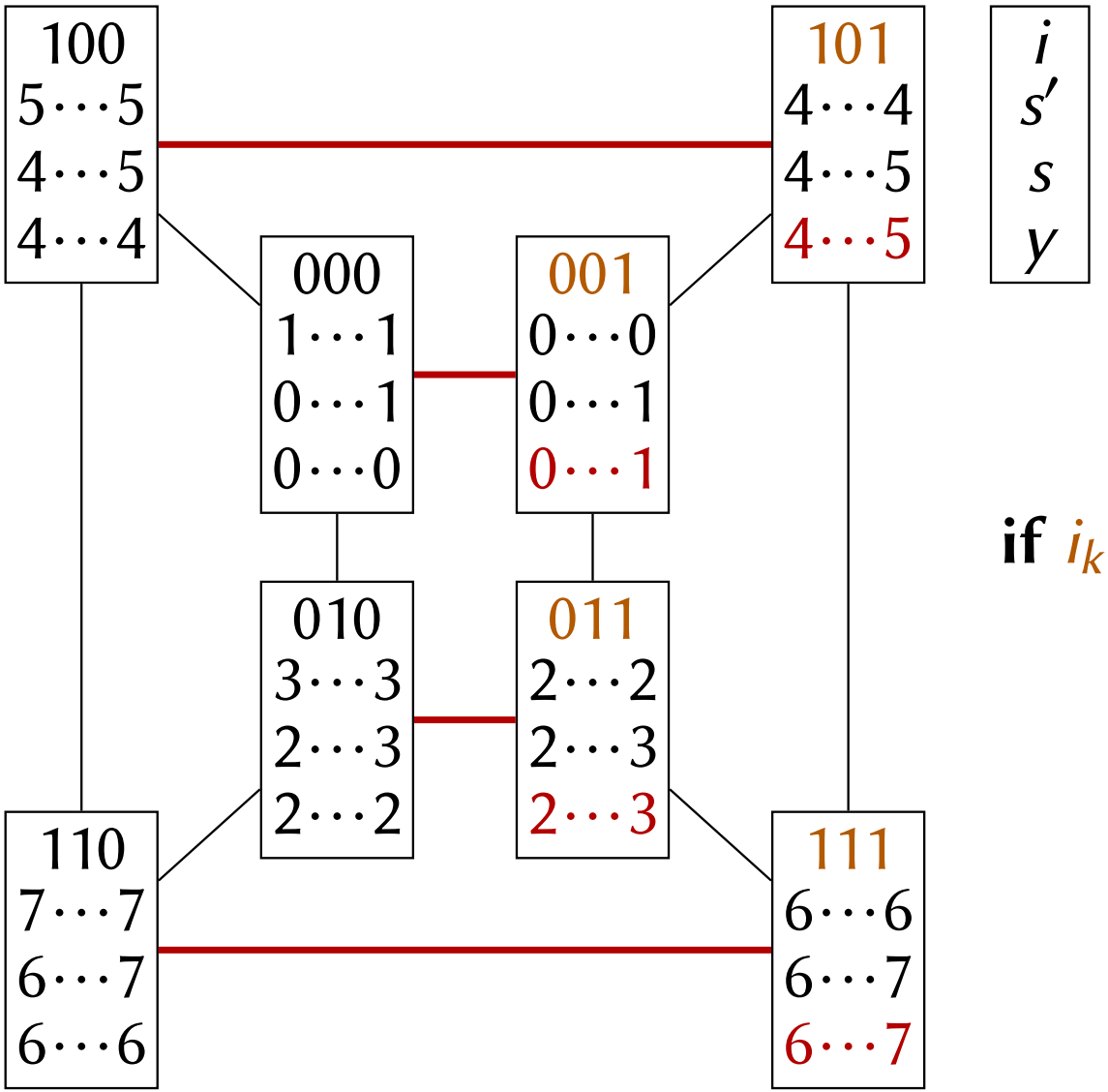
Präfixsummen: Hyperwürfel-Algorithmus ($k = 0$)



$s \leftarrow s \otimes s'$

$k \cdots m = \bigotimes_{j=k}^{m} x_j$

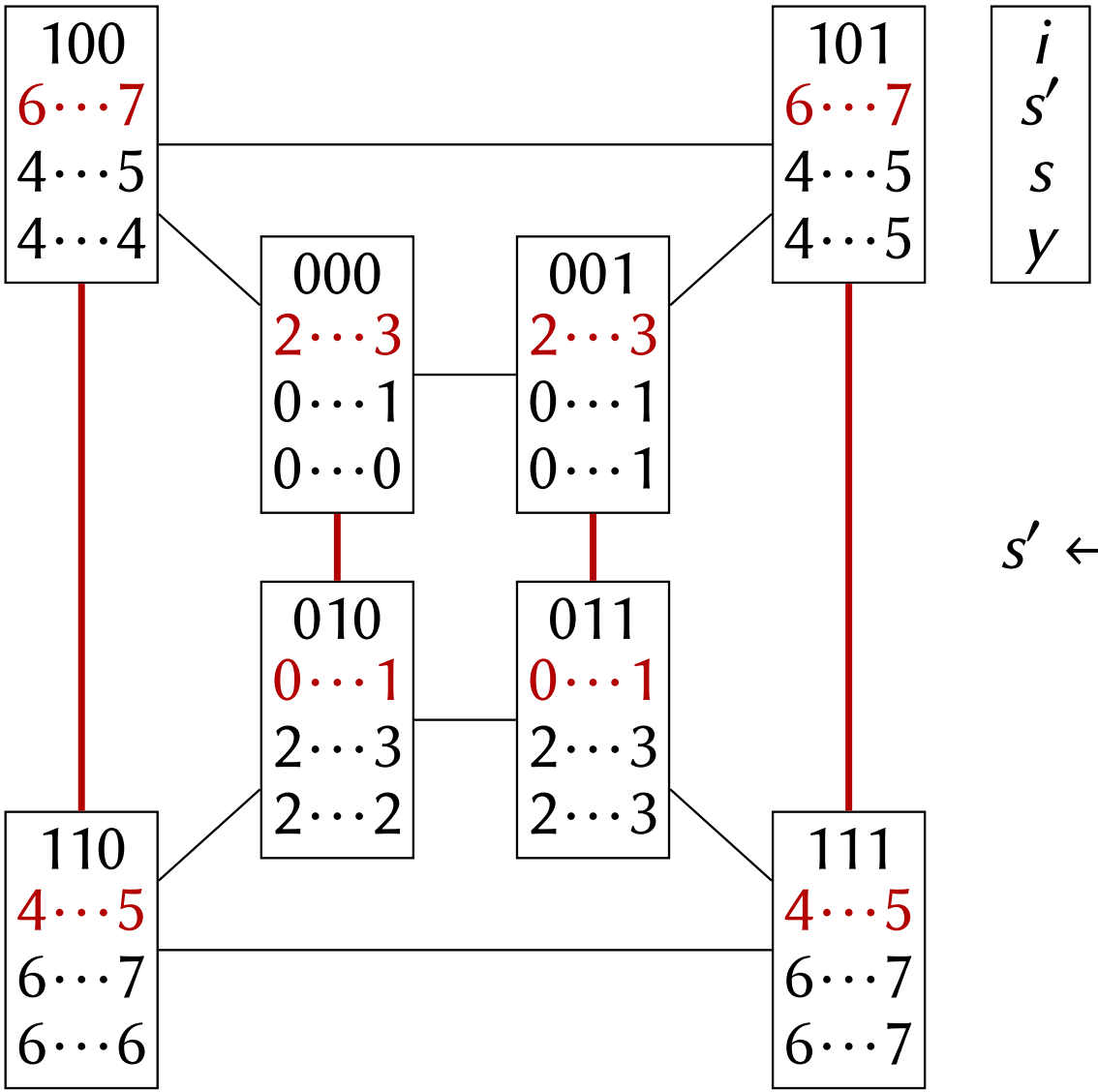
Präfixsummen: Hyperwürfel-Algorithmus ($k = 0$)



if $i_k = 1$ **then** $y \leftarrow y \otimes s'$

$$k \cdots m = \bigotimes_{j=k}^{m} x_j$$

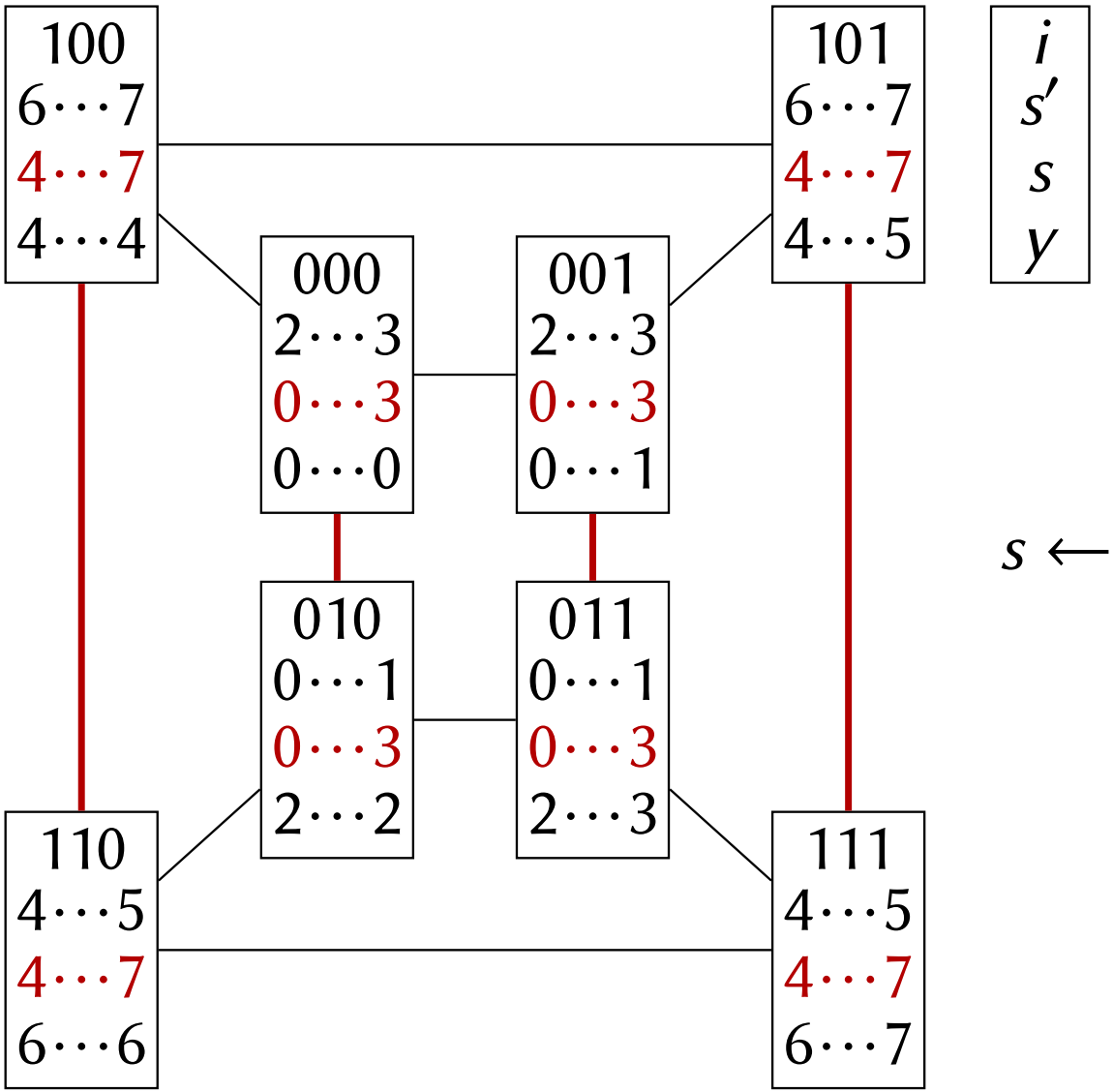
Präfixsummen: Hyperwürfel-Algorithmus ($k = 1$)



$s' \leftarrow \text{SENDRECV}(s, i \oplus 2^1, i \oplus 2^1)$

$k \cdots m = \bigotimes_{j=k}^{m} x_j$

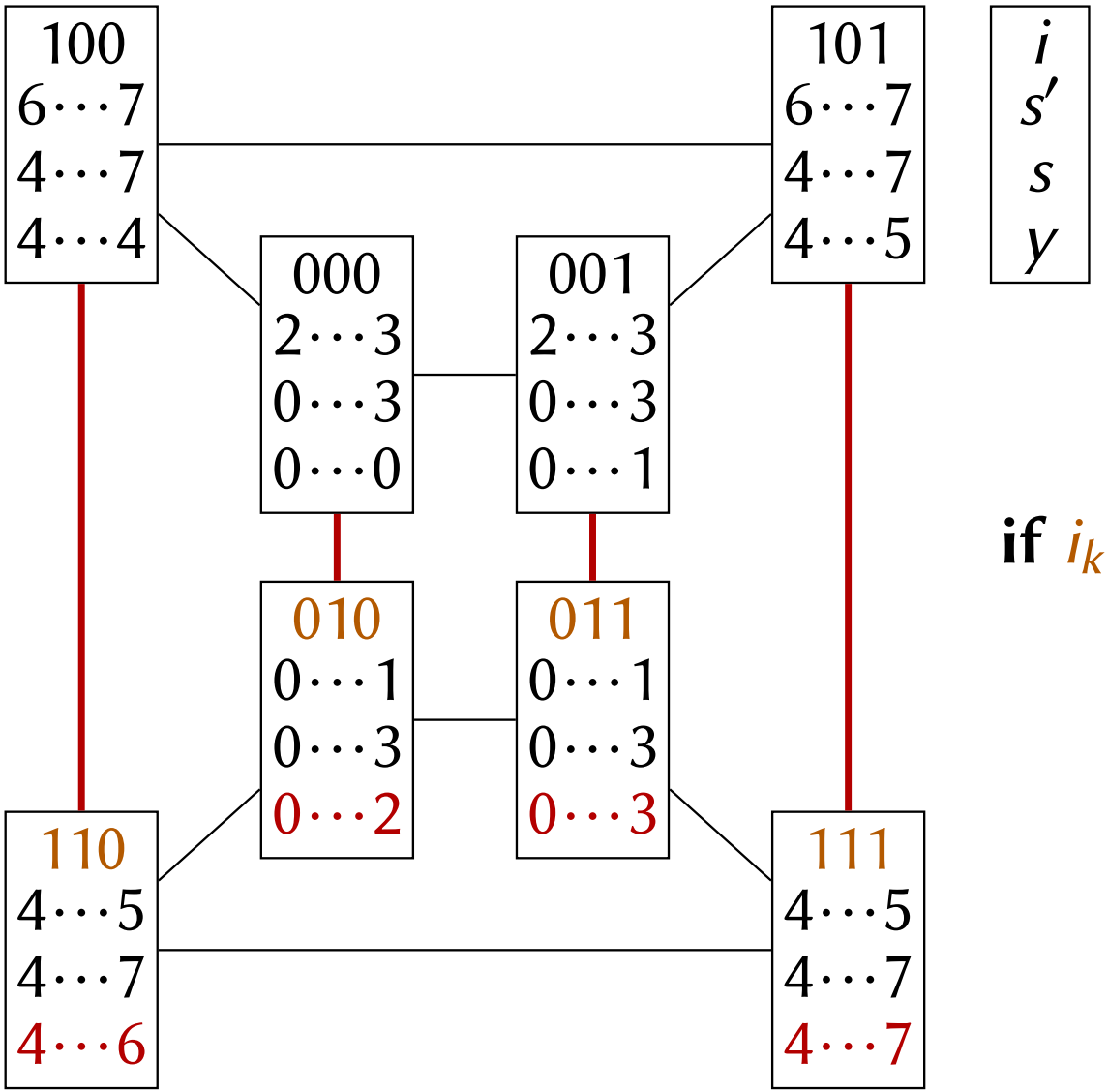
Präfixsummen: Hyperwürfel-Algorithmus ($k = 1$)



$$s \leftarrow s \otimes s'$$

$$k \cdots m = \bigotimes_{j=k}^{m} x_j$$

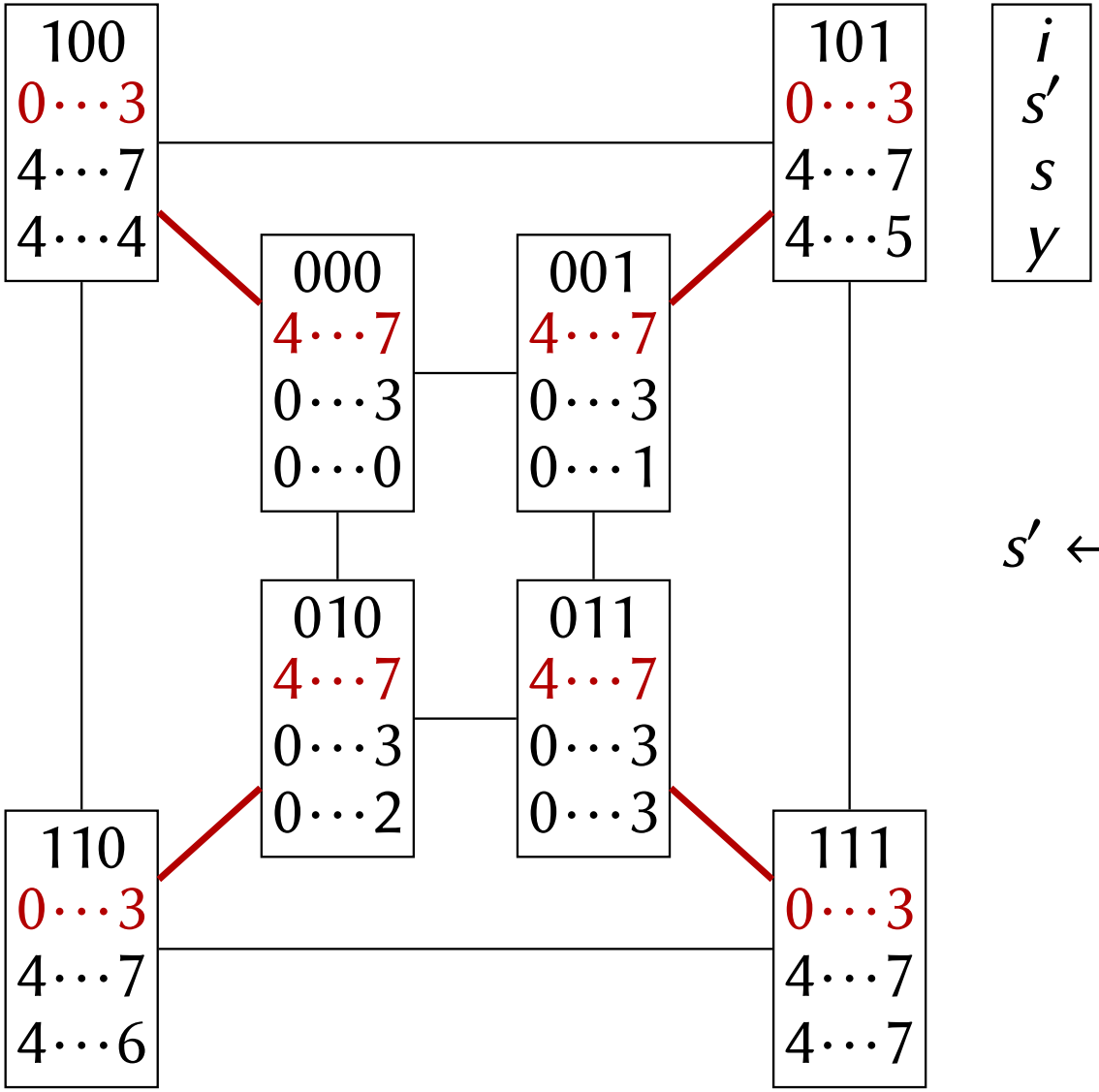
Präfixsummen: Hyperwürfel-Algorithmus ($k = 1$)



if $i_k = 1$ **then** $y \leftarrow y \otimes s'$

$$k \cdots m = \bigotimes_{j=k}^{m} x_j$$

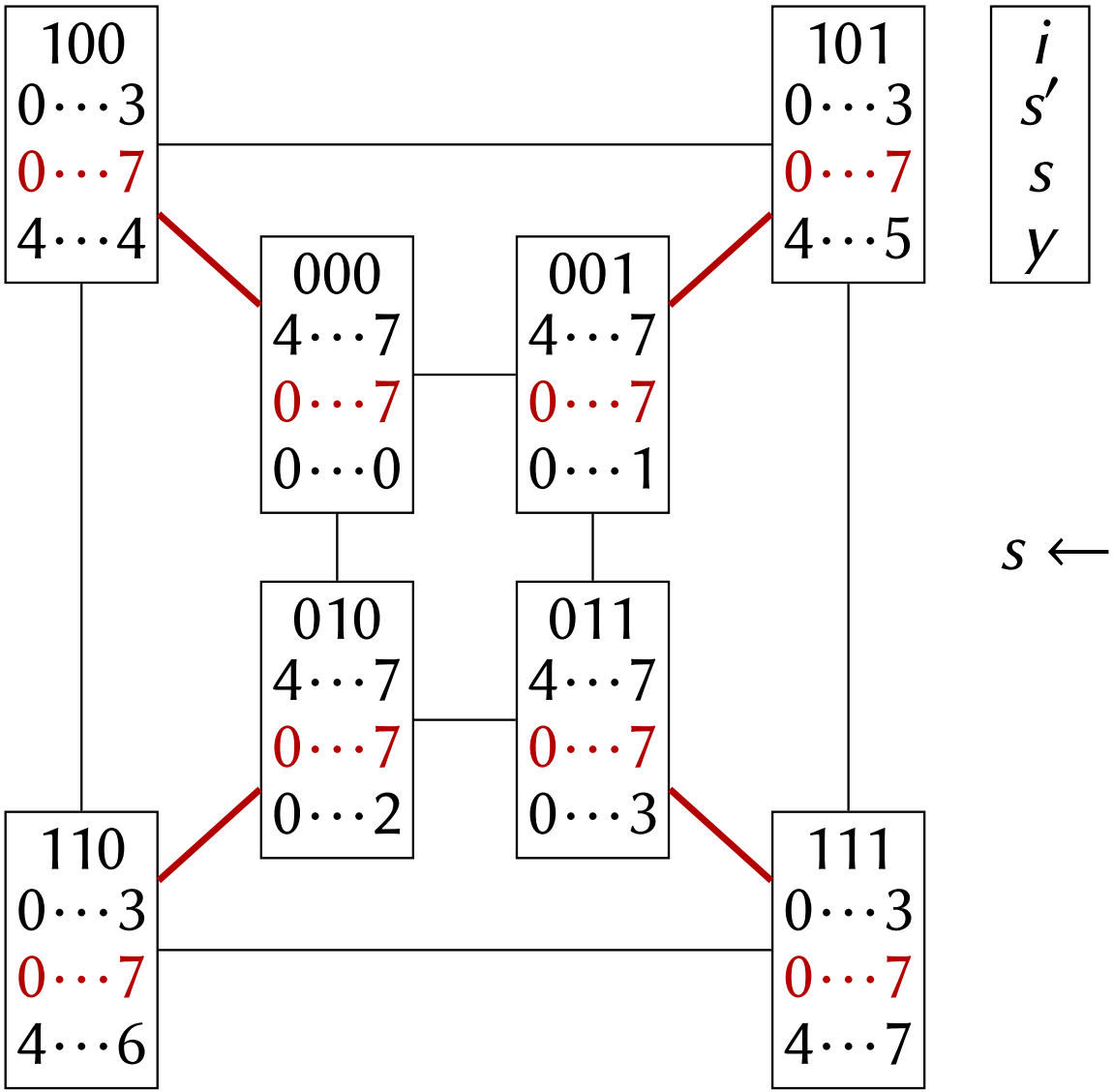
Präfixsummen: Hyperwürfel-Algorithmus ($k = 2$)



$$s' \leftarrow \text{SENDRECV}(s, i \oplus 2^2, i \oplus 2^2)$$

$$k \cdots m = \bigotimes_{j=k}^{m} x_j$$

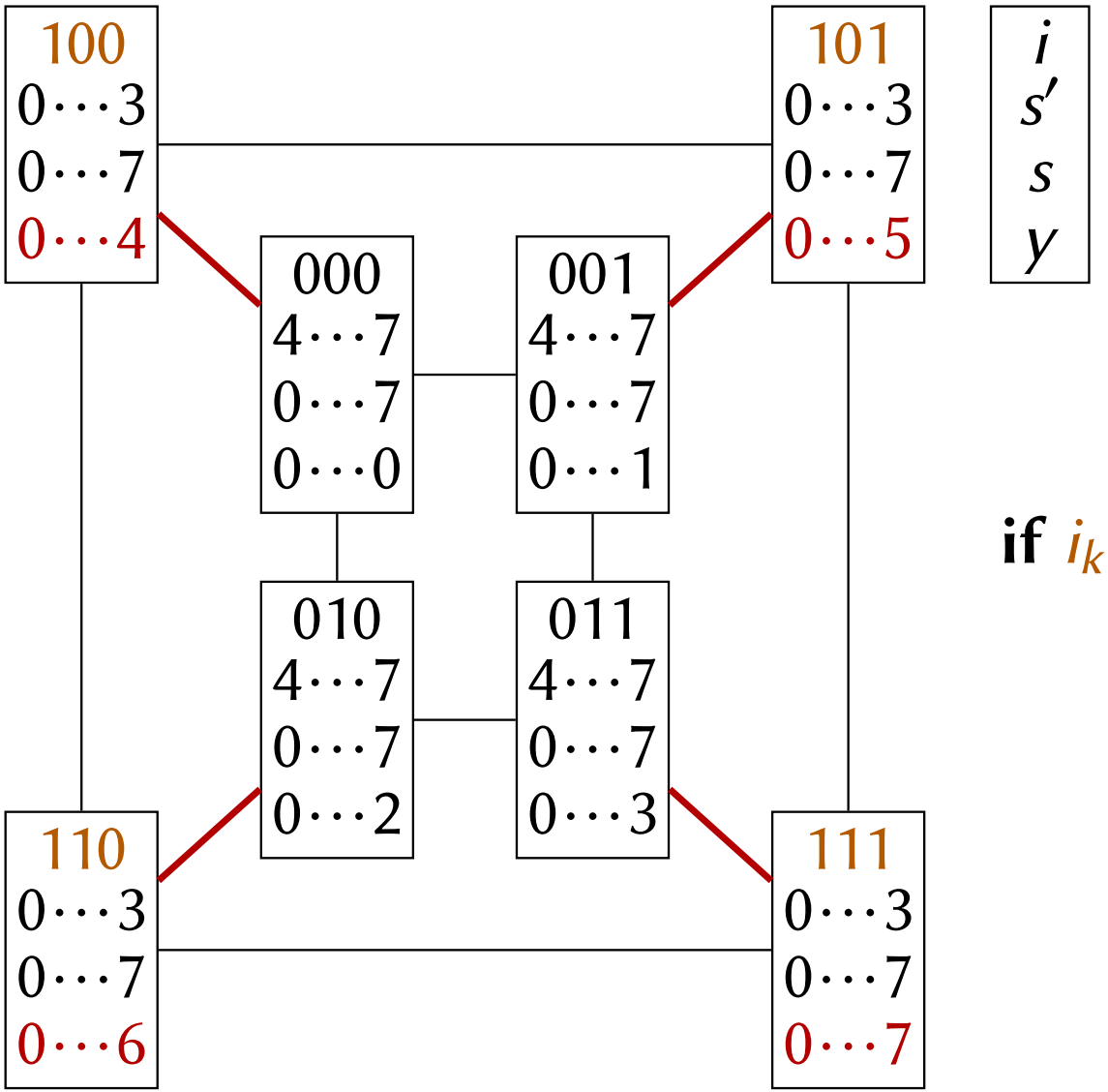
Präfixsummen: Hyperwürfel-Algorithmus ($k = 2$)



$$s \leftarrow s \otimes s'$$

$$k \cdots m = \bigotimes_{j=k}^{m} x_j$$

Präfixsummen: Hyperwürfel-Algorithmus ($k = 2$)



if $i_k = 1$ **then** $y \leftarrow y \otimes s'$

$$k \cdots m = \bigotimes_{j=k}^{m} x_j$$

Hyperwürfel-Algorithmus: Korrektheit

schreibe $i|0^k$ für $(i_{d-1}, \ldots, i_k, 0, \ldots, 0)$ und
analog $i|1^k$ für $(i_{d-1}, \ldots, i_k, 1, \ldots, 1)$

1 $y \leftarrow x$ // auf PE i liegt x_i
2 $s \leftarrow x$ // für Summe von Elementen in Unterwürfel
3 **for** $k \leftarrow 0$ **to** $d-1$
// Invariante: $s_i = \bigotimes_{a_i}^{b_i} x_i$ und $y_i = \bigotimes_{a_i}^{i} x_i$
// mit $a_i = i|0^k$, $b_i = i|1^k$
4 $s' \leftarrow \text{SENDRECV}(s, i \oplus 2^k, i \oplus 2^k)$
5 $s \leftarrow s \otimes s'$
6 **if** $i_k = 1$
7 $y \leftarrow y \otimes s'$
8 // auf PE i liegt $y_i = x_0 \otimes \cdots \otimes x_i$

Hyperwürfel-Algorithmus: Analyse

- Laufzeit

$$T_{prefix} \in O((T_{start} + \ell \cdot T_{byte}) \cdot \log p)$$

- für $T_{start} \ll \ell T_{byte}$ nicht optimal
  - $\Theta(T_{start} \cdot \log p + \ell \cdot T_{byte})$ erreichbar
  - $\rightsquigarrow$ Vorlesung «Parallele Algorithmen»

- analog übrigens auch schon bei Reduktion

Parallele Präfixsummen für nicht-kommutatives $\otimes$

kleine Korrekturen an zwei Stellen:

1 $y \leftarrow x$ // auf PE i liegt x_i
2 $s \leftarrow x$ // für Summe von Elementen in Unterwürfel
3 **for** $k \leftarrow 0$ **to** $d-1$
4 $s' \leftarrow \text{SENDRECV}(s, i \oplus 2^k, i \oplus 2^k)$
// statt $s \leftarrow s \otimes s'$
5 **if** $i_k = 1$
6 $s \leftarrow s' \otimes s$
7 **else** $s \leftarrow s \otimes s'$
8 **if** $i_k = 1$
9 $y \leftarrow s' \otimes y$ // statt $y \otimes s'$
10 // auf PE i liegt $y_i = x_0 \otimes \cdots \otimes x_i$

Parallele Präfixsummen für nicht-kommutatives $\otimes$

- $x \otimes y = x$: parallele Präfixsumme *eine* Möglichkeit
  - in Zeit $\log p$
  - x_0 an alle Prozessoren zu verteilen
- *Broadcast*
  - $rval \leftarrow \textsc{bcast}(val, from)$
  - Anwendung «wie in MPI»
  - muss von *allen* Prozessoren aufgerufen werden

Überblick

Aufgabenvarianten

wo liegen am Anfang die Daten

- alle n Elemente auf PE 0
  - jedes Element von PE 0 mindestens einmal angefasst
  - $\rightsquigarrow$ Laufzeit in $\Omega(n)$
- je n/p Elemente auf PE i, $\quad 0 \leq i < p$
  - interessanter

Verteilte Eingabedaten

- zunächst einfacher Fall $p = n$
  - Prozessor i hat Eingabeelement x_i
- Grundidee von Quicksort beibehalten
  - ein Element pv als Pivot wählen
  - Elemente umverteilen
    - Elemente kleiner als pv auf Prozessoren mit kleinen Rängen
    - Elemente größer als pv auf Prozessoren mit großen Rängen
  - parallele Rekursion: gleichzeitig
    - Elemente kleiner als pv rekursiv analog und
    - Elemente größer als pv rekursiv analog sortieren

Theoretiker-Quicksort

Theoretiker-Quicksort Teil 0: Vorbereitungen

theoQSort0(x)

```
1  i ← commRank()
2  p ← commSize()
   // PE i hat Element x_i
3  theoQSort(x, 0, p − 1)
```

Theoretiker-Quicksort

Theoretiker-Quicksort Teil 1: kleine Elemente zählen

THEOQSORT$(x, \ldots)$

1 **if** $i = 0$
2 $\quad ipv \leftarrow$ RANDINT$(0, p)$ // expected log. recursion levels
3 $ipv \leftarrow$ BCAST$(ipv, 0)$
4 $pv \leftarrow$ BCAST(x, ipv)
5 $small \leftarrow x \leq pv$ // 1 iff $x \leq pv$; 0 otherwise
6 $j \leftarrow$ PREFIXSUM$(small, +)$
7 $p' \leftarrow$ BCAST$(j, p-1)$ // nr of elements $\leq pv$

Theoretiker-Quicksort

Theoretiker-Quicksort Teil 2: Datenumverteilung und Rekursion

```
    small ← x ≤ pv                          // 1 iff x ≤ pv; 0 otherwise
    j ← PREFIXSUM(small, +)                 // «inclusive» prefix sum
                                            // as described before
    p' ← BCAST(j, p − 1)                    // nr of elements ≤ pv
1   if small = 1
2       SEND(x, j − 1)                      // assume nonblocking SEND
3   else SEND(x, p' + i − j)                // assume nonblocking SEND
                                            // i − j small elements left of i
4   x ← RECV(ANY)
5   recursive THEOQSORT of «left»/«right» part
```

Theoretiker-Quicksort: Laufzeit

- erwartete Rekursionstiefe $O(\log p)$
- jeweils Zeit $O(T_{start} \log p)$
- insgesamt erwartete Zeit $O(T_{start}(\log p)^2)$