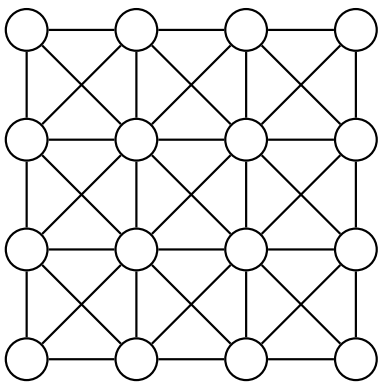
4 Parallelisierung durch Datenzerlegung

4.1 Zellularautomat $(Q, 0, N, \delta, d)$

- Endliche Zustandsmenge Q
- Ruhezustand 0
- Nachbarschaft (Raster) N. Zum Beispiel M_1: Eine Zelle und ihre 8 Nachbarn in horizontaler und diagonaler Richtung. $(c, n, o, s, w, no, so, sw, nw)$
- Überführungsfunktion $\delta : Q^{|N|} \to Q$.
- Dimension: 1, 2, 3, (4), ...



Arbeitsweise

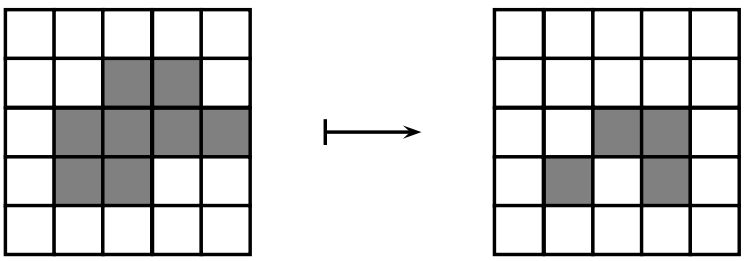
- synchron agierende
- identische
- kommunizierende
- endliche Automaten

In einem Zeitschritt führen alle endlichen Automaten synchron einen Zustandsübergang aus.

4.1.1 Eine (Fast-)Majoritätsregel

- $Q = \{0, 1\}$
- $d = 2$
- $N = M_1$ Moore-Nachbarschaft
- $\delta(c, n, o, s, w, no, so, sw, nw) =$
  $f(c + n + o + s + w + no + so + sw + nw)$

n	0	1	2	3	4	5	6	7	8	9
$f(n)$	0	0	0	0	1	0	1	1	1	1



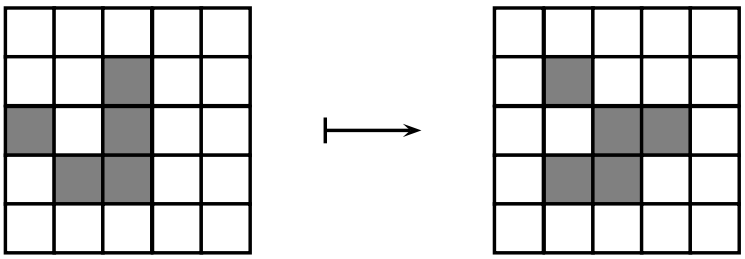
Ein Modell für “Entmischung”

Huebsche Bilder die der Drucker nicht mag...

LIFE

$$\delta(q_{\mathbf{c}}, q_{\mathbf{n}}, q_{\mathbf{o}}, q_{\mathbf{s}}, q_{\mathbf{w}}, q_{\mathbf{no}}, q_{\mathbf{so}}, q_{\mathbf{sw}}, q_{\mathbf{nw}})$$
$$= g(q_{\mathbf{c}}, q_{\mathbf{n}} + q_{\mathbf{o}} + q_{\mathbf{s}} + q_{\mathbf{w}} + q_{\mathbf{no}} + q_{\mathbf{so}} + q_{\mathbf{sw}} + q_{\mathbf{nw}}) \ .$$

n	0	1	2	3	4	5	6	7	8
$g(0,n)$	0	0	0	1	0	0	0	0	0
$g(1,n)$	0	0	1	1	0	0	0	0	0



Sequentielle Simulation

4.1.2 Eine einfache Version

caseq-slow.c

```
/* simulate a cellular automaton
 * (sequential version)
 * periodic boundaries
 *
 * uses graphical display
 *
 * command line arguments:
 *
 * #1: number of rows of configurations;
 * #2: number of iterations to be simulated;
 * #3: length of period (in iterations) between
 *     displaying configurations;
 *     0 means that no states are displayed;
 * #4: name of X Display; e.g. "mpii01666:0.0";
 */
```

Datentypen

```
#define CONFIGCOLS 256

typedef char State;
typedef State Row[CONFIGCOLS + 2];
```

Idee: Flexibleres C++ Programm? Effizienz?

Überführungsfunktion

```
/* transition table for the annealing rule */
static State anneal[10] =
   {0, 0, 0, 0, 1, 0, 1, 1, 1, 1};

/*  compute the new state of a cell */
#define transition(a, x, y)
   (anneal[(a)[(y)-1][(x)-1] + \
           (a)[(y)  ][(x)-1] + \
           (a)[(y)+1][(x)-1] + \
           (a)[(y)-1][(x)  ] + \
           (a)[(y)  ][(x)  ] + \
           (a)[(y)+1][(x)  ] + \
           (a)[(y)-1][(x)+1] + \
           (a)[(y)  ][(x)+1] + \
           (a)[(y)+1][(x)+1]   \
          ])
```

Idee: C++ mit `inline` Funktion.

Simulation

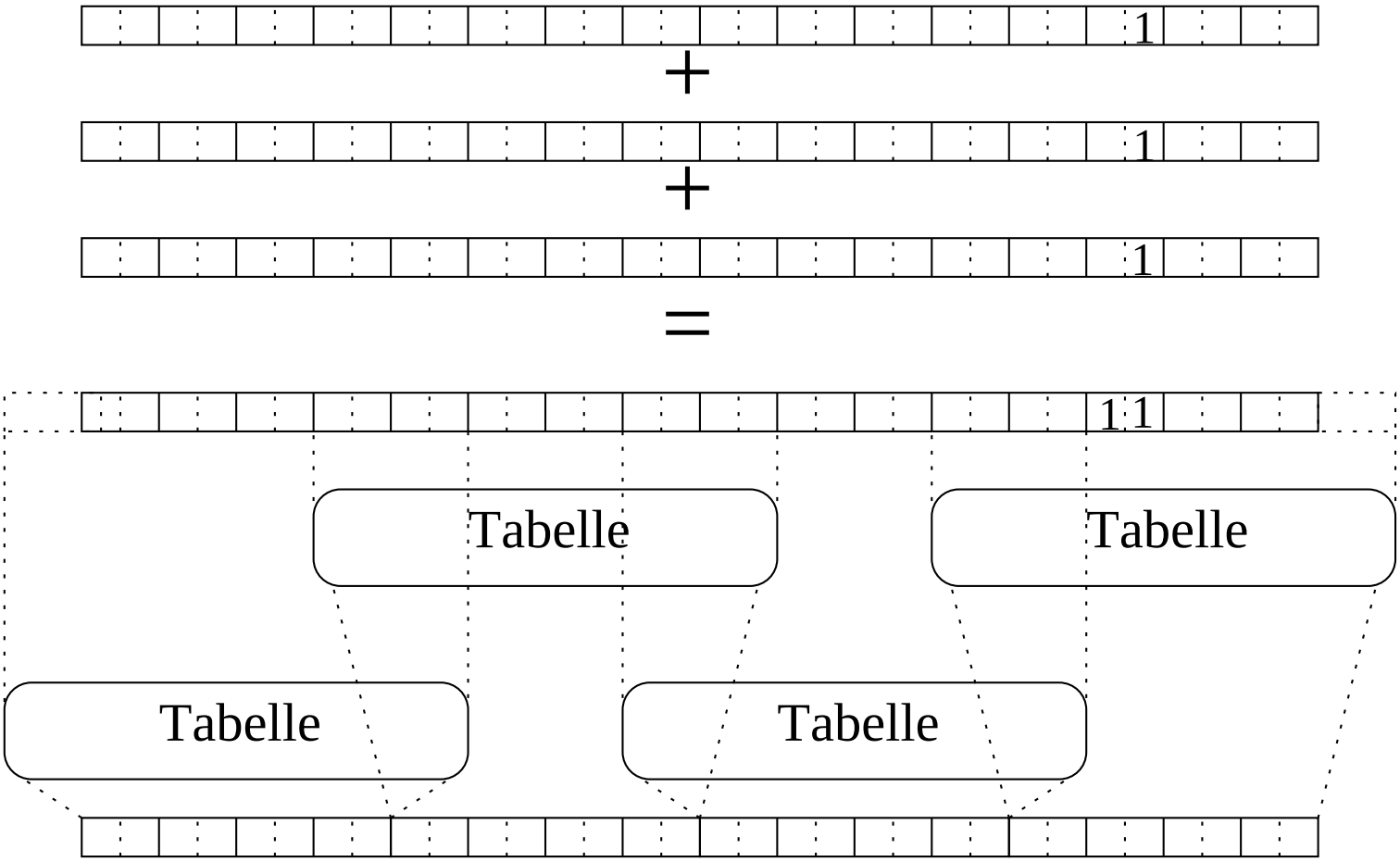
```
/*  simulate one step of the CA
 *  IN:
 *  from:       pointer to old configuration
 *  to:         pointer to new configuration
 *  configRows: number of rows of configurations
 *  OUT:
 *  to:         the memory is modified
 */
static void
simulate(Row *from, Row *to, int configRows)
{
   int x,y;

   boundary(from, configRows);
   for (y = 1;  y <= configRows;  y++) {
      for (x = 1;  x <= CONFIGCOLS;  x++) {
         to[y][x  ] = transition(from, x  , y);
      }
   }
}
```

Periodische Randbedingungen

```
/*  treat boundary conditions
 *  IN:
 *  buf:       pointer to a stored configuration
 *  configRows:number of rows of configurations
 *  OUT:
 *  buf:       configuration is modified in place
 */
static void boundary(Row *buf, int configRows)
{ int x,y;

  for (y = 1;  y <= configRows;  y++) {
    buf[y][0             ] = buf[y][CONFIGCOLS];
    buf[y][CONFIGCOLS+1] = buf[y][1          ];
  }

  for (x = 1;  x <= CONFIGCOLS;  x++) {
    buf[0             ][x] = buf[configRows][x];
    buf[configRows+1][x] = buf[1          ][x];
  }
}
```

INFORMATIK

4.1.3 Bitparallele Simulation



Vergleich

- Parallele Addition für 16 Zellen
- Paralleler Lookup für 4 Zellen
- 4-fach geringerer Speicherbedarf (Cache?)
- 4-fach mehr Zellen pro Cache-miss
- 16-fach mehr Zellen pro Load- (Store-) Befehl
- Pentium-II-300: 20M Zellen/s $\Longrightarrow$ 160M Zellen/s (512×512 Zellen)

Umsetzung

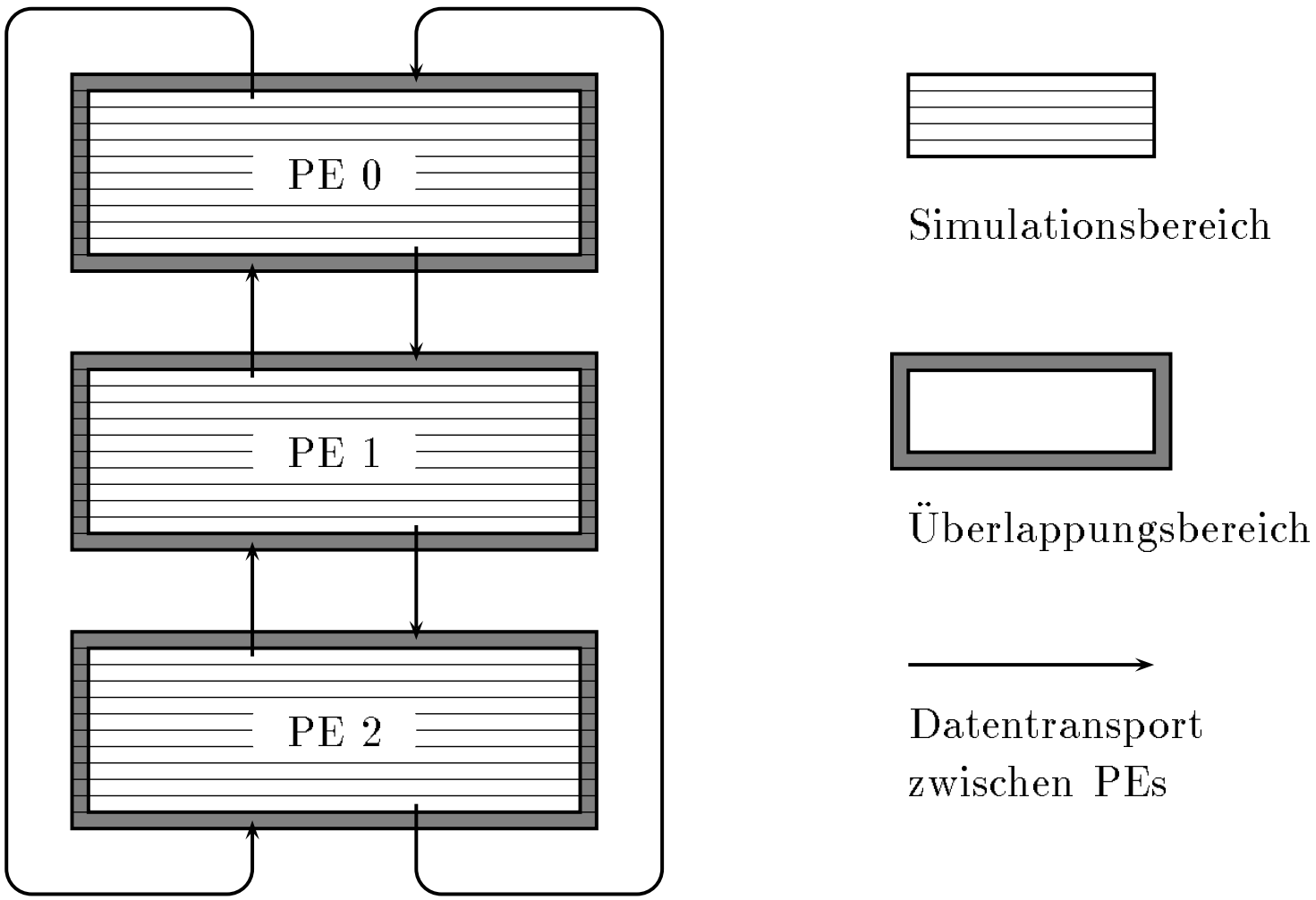
```
static void simulate(Line *from, Line *to, int lines)
{ int x, y;
  unsigned int l, m, r;
  boundary(from, lines);
  for (y = 1;  y <= lines;  y++) {
    l = from[y-1][0] + from[y][0] + from[y+1][0];
    m = from[y-1][1] + from[y][1] + from[y+1][1];

    for (x = 1;  x <= XSIZE; x++) {
      r=from[y-1][x+1]+from[y][x+1]+from[y+1][x+1];
      to[y][x] =
        (transition[((l& 3u        )<<10)|
                    ((m&(1023u<<22))>>22) ]<<24)|
        (transition[((m&(4095u<<14))>>14) ]<<16)|
        (transition[((m&(4095u<< 6))>> 6) ]<< 8)|
         transition[((m& 1023u     )<< 2)|
                    ((r&(   3u<<30))>>30)  ];
      l = m; m = r;
    }
  }
}
```

Weitere Maßnahmen:

- 64 Bit
- Paralleler Lookup für 8 Zellen. Wirklich?
- 1 Bit pro Zustand
  - $-$ δ als boolesche Funktion `& | ^ ~`. (LIFE)
  - $+$ 2–16 mal mehr Parallelismus
  - $+$ Keine Tabellenzugriffe
  - $-$ Viele Instruktionen
- Ausrichtung nach Cache-Line-Grenzen
- Prefetching, Zeilenpuffer
- MMX, Grafikhardware?
- Einfachere Behandlung stationärer Bereiche. Datenstrukturen ...

4.2 Parallelisierung durch Datenzerlegung

4.2.1 Streifenweise Zerlegung



Ein einfacher Ansatz:

- Streifenweise (anderes folgt)
- Alle PEs gleich viel. Zwei Bedingungen?
- Nur Randbehandlung wird geändert

4.2.2 MPI_Sendrecv

```
int MPI_Send(
  void *buf, int count, MPI_Datatype datatype,
  int dest, int tag,
  MPI_Comm comm);
```

$+$

```
int MPI_Recv(
  void *buf, int count, MPI_Datatype datatype,
  int source, int tag,
  MPI_Comm comm, MPI_Status *status);
```

$\leadsto$

```
int MPI_Sendrecv(
  void *sbuf, int scount, MPI_Datatype stype,
  int dest, int stag,
  void *rbuf, int rcount, MPI_Datatype rtype,
  int source, int rtag,
  MPI_Comm comm, MPI_Status *status);
```

Neue Randbehandlung

```
#define STATE MPI_CHAR

static void yBoundary(Line *buf, int lines)
{ MPI_Status dummy;

  /* send down */
  MPI_Sendrecv(buf[lines], XSIZE + 2, STATE,
               (iProc + 1         ) % nProc, 42,
               buf[0    ], XSIZE + 2, STATE,
               (iProc - 1 + nProc) % nProc, 42,
               MPI_COMM_WORLD, &dummy);

  /* send up */
  MPI_Sendrecv(buf[1]          , XSIZE + 2, STATE,
               (iProc - 1 + nProc) % nProc, 43,
               buf[lines + 1], XSIZE + 2, STATE,
               (iProc + 1         ) % nProc, 43,
               MPI_COMM_WORLD, &dummy);
}
```

4.2.3 Ergebnisse Einsammeln

```
int MPI_Gather(
  void *sendbuf, int sendcount, MPI_Datatype send
  void *recvbuf, int recvcount, MPI_Datatype recv
  int root, MPI_Comm comm);
```

- PE `root` empfängt P Nachrichten nach `recvbuf`
- **Alle** PEs senden Nachricht der Länge `recvcount`
- Alle Werte für `root` gleich
- `recvcount` $= P \cdot$ `sendcount` (modulo Datentypmagie)
- `recvxxx` nur auf root beachtet.
- Teuer (Warum?)
- `MPI_Gatherv` Verallgemeinerung (Fortran-mäßig)

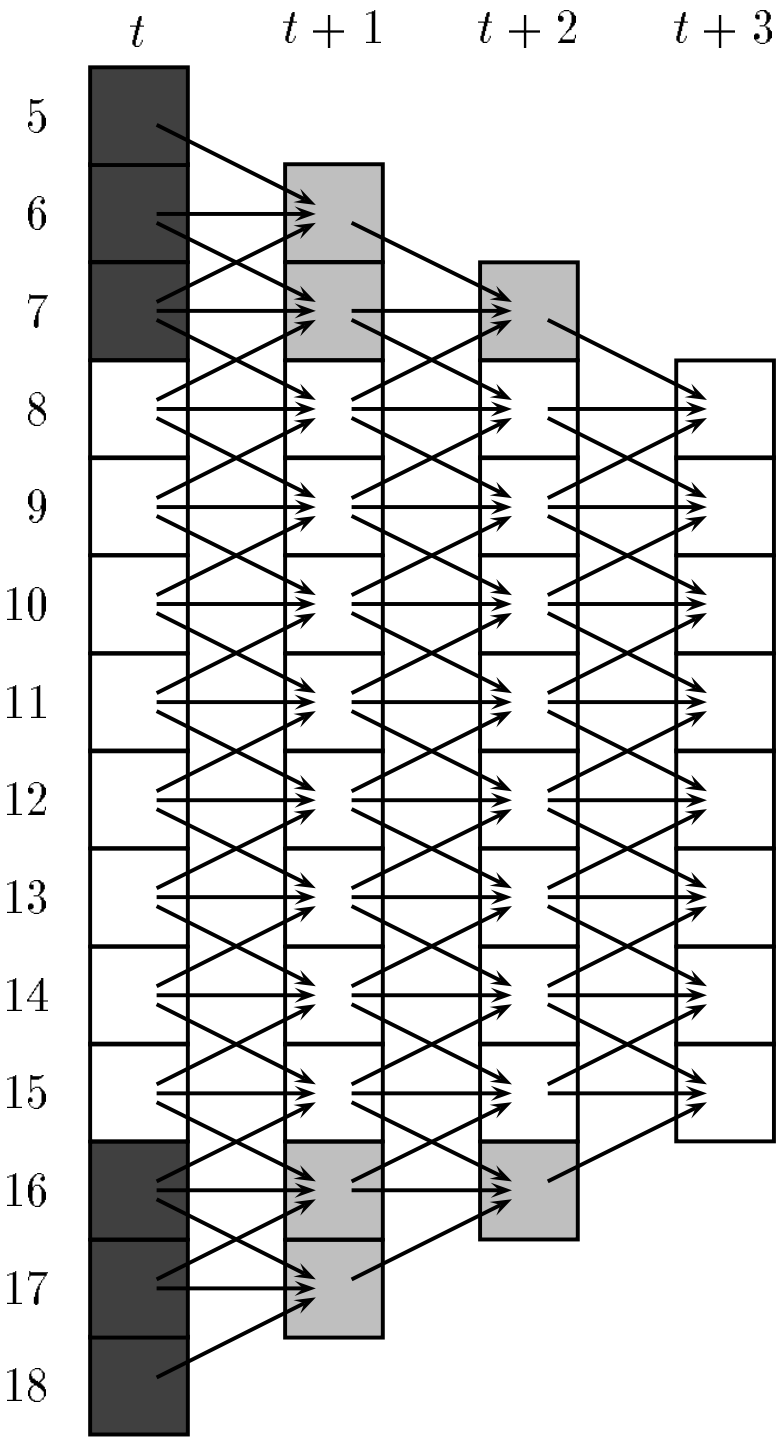
Für Simlator: $P|$Zeilenzahl vereinfacht Dinge

Analyse

Zeit pro Iteration:

- $T_{\text{seq}} \approx n^2 T_{\text{sim}}$
- $T_{\text{par}} \approx \frac{n^2}{P} T_{\text{sim}} + 4(T_{\text{start}} + \frac{n}{4} T_{\text{byte}})$
- $T_{\text{sim}} \approx 6 \cdot 10^{-9}\text{s}$
- $T_{\text{start}} \approx 300000 \cdot 10^{-9}\text{s}$
- $T_{\text{byte}} \approx 30 \cdot 10^{-9}\text{s}$
- $E = \frac{T_{\text{seq}}}{P T_{\text{par}}} > 1/2$???

4.2.4 Entkopplung von Berechnungen



von Nachbar-PE importierter Zustand

von PE redundant berechneter Zustand

von PE berechneter Zustand aus seinem eigenen Zuständigkeitsbereich

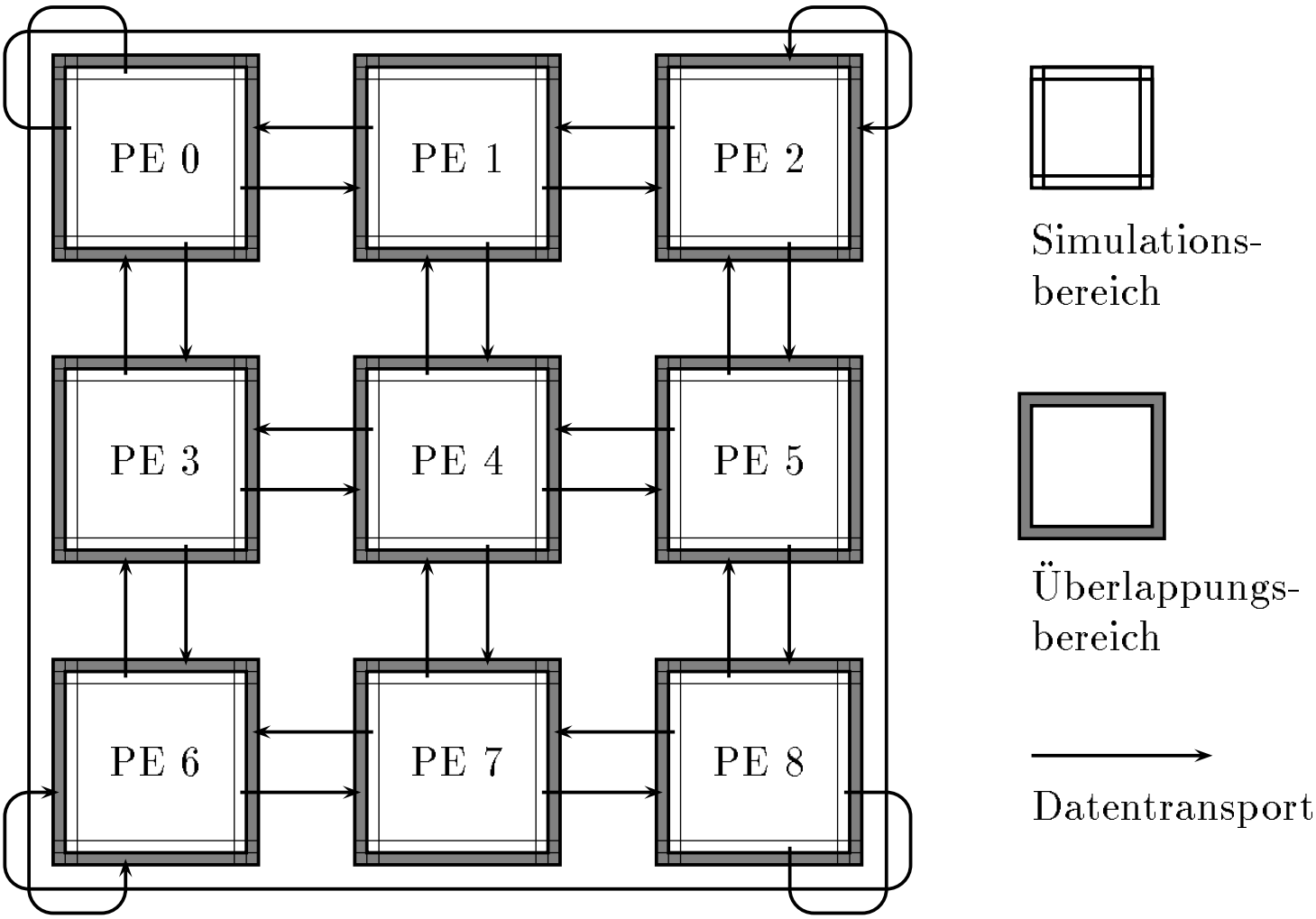
Datenabhängigkeit

4.2.5 Analyse

- Überlappungsgröße k
- k Iterationen:
  $T_k = kT_{\text{sim}}(m+k-1)n + 4(T_{\text{start}} + \frac{n}{4}kT_{\text{byte}})$
- $kT_{\text{par}} - T_k = kT_1 - T_k =$
- $(k-1)(4T_{\text{start}} - knT_{\text{sim}})$
- Ab $2T_{\text{start}} > nT_{\text{sim}}$ lohnt sich $k \geq 2$
- Welches k ist optimal? Minimiere T_k/k
- D.h. $\frac{\partial}{\partial k}T_{\text{neu}} = 0$
- $k_{\text{opt}} = 2\sqrt{\frac{T_{\text{start}}}{nT_{\text{sim}}}}$
- Wann wird jetzt gute Effizienz erreicht?

INFORMATIK

Zweidimensionale Problemzerlegung



Diskussion

- $T_{\text{par}} \approx \frac{n^2}{P} T_{\text{sim}} + \mathbf{8}(T_{\text{start}} + \frac{n}{4\sqrt{\mathbf{P}}} T_{\text{byte}})$
- Gut wenn $nT_{\text{byte}} \gg T_{\text{start}}$

\+ Geringer Kommunikationsumfang vor allem bei großem P.

\+ Weniger redundante Arbeit bei Entkopplung

– „Magische Prozessorzahlen“

– Spaltenaustausch ist kompliziert (MPI Datentypen helfen etwas, Entkopplung hilft)

- Nichquadratische Teile gut?

Übungsumgebung

- Solarix x86
- Maschinen: `orwell`x, $x \in \{1, \ldots, 16\}$. Später: einloggen auf `bigbrother`
- `.tcshrc_private:`

```
setenv PATH ${PATH}:/opt/mpi-1.0.13/bin
setenv PATH ${PATH}:/opt/mpi-1.0.13/lib/i86pc/ch_p4
setenv PATH ${PATH}:/KM/usr/sanders/praktikum/bin
```

- rhosts file
- Übersetzen: `pcc, pgcc, pg++`
- Programm starten:
  `mpirun -np` P `prg params`
  siehe auch `mpich` Handbuch