INFORMATIK

# Vorlesung + Übung Parallele Programmierung

**Dozent:** Peter Sanders

**Umfang:** 2 + 2 SWS

**Vorlesung:** Do 14-16, SR 013, Geb. 45. (21.5., 11.6: Feiertag)

**Übung:** ???



INFORMATIK

# Literatur

- P. Sanders und T. Worsch, „Parallele Programmierung mit MPI – ein Praktikum“, ISBN 3-931216-76-4, Logos Verlag Berlin, 1997.
- V. Kumar, A. Grama, A. Gupta und G. Karypis, „Introduction to Parallel Computing. Design and Analysis of Algorithms“, Benjamin/Cummings, 1994.
- `http://www.mpi-sb.mpg.de/~sanders/parprog.html`
  - MPI Forum, „MPI: A Message-Passing Interface Standard“, `http://www.mpi-forum.org/docs/mpi-11.ps.Z`.
  - Folien
  - u.v.a.m.



INFORMATIK

# Überblick

1. Parallele Maschinen und Modelle
2. Routingalgorithmen, Kostenmodelle, MPI Anfangsgründe
3. Zellularautomaten, Parallelisierunge duch Datenzerlegung
4. Latenzzeitverdeckung, Entkopplung, MPI Punkt-zu-Punktkommunikation
5. Parallele Matrizenrechnung, MPI kollektive Operationen
6. Lineare Gleichungssysteme, PDE, FFT
7. Sortieren

8./9. Lastverteilung I + II

10. MPI-2, Shared Memory
11. Komplexere Anwendungen



INFORMATIK

# 1 Aufbau von Parallelrechnern

## 1.1 Einzelne Prozessoren

### 1.1.1 Befehlspipeline

(siehe auch: J.L. Hennessy, D. A. Patterson, „Computer Architecture a Quantitative Approach“, 2. Auflage, 1996)

T. Ungerer, Mikroprozessortechnik, „TAT-Verlag“, 1995

### 1.1.2 Befehlsparallelismus

- Initiiere mehrere (z.B. 4) Befehle pro Takt (superskalar, VLIW, ... )
- Mehrere parallele Funktionseinheiten (z.T. mit Pipeline)
- Geht nur wenn Befehle sich nicht „in die Quere“ kommen.
- Sprungvorhersage wichtig
- Gute Compiler wichtig
- Trotzdem bleibt effiziente Programmierung schwer



### 1.1.3 Speicherhierarchie

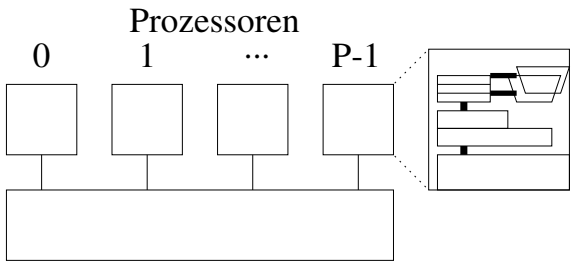
Ein (typischer?) 4-fach superskalarer 500MHz RISC-Prozessor.

| | Verzögerung [s] | Bandbreite [Byte/s] | Kapazität [Byte] |
|---|---|---|---|
| Register | $2 \cdot 10^{-9}$ | $64 \cdot 10^{9}$ | 512 |
| L-1 Cache | $4 \cdot 10^{-9}$ | $8 \cdot 10^{9}$ | 2 x 8–32K |
| L-2 Cache | $20 \cdot 10^{-9}$ | $2 \cdot 10^{9}$ | 512K–4M |
| DRAM | $400 \cdot 10^{-9}$ | $500 \cdot 10^{6}$ | 64M – 1G |



## 1.2 Typen von Parallelrechnern

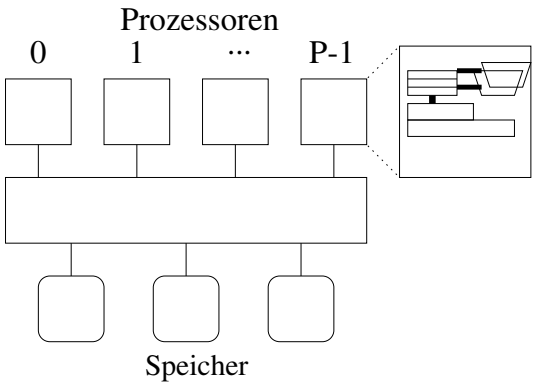
### 1.2.1 Distributed Memory




- Sei $P$ Anzahl Prozessoren (PE=Processing Element).
- PEs sind von 0 bis $P-1$ durchnumeriert



### 1.2.2 (Symmetric) Shared Memory




- Beispiel: Sun E10000
- Kann durch distributed memory emuliert werden indem Zugriffe auf nichtlokalen Speicher in Netzwerkkommunikationen umgesetzt werden. (Zum Beispiel SGI Origin2000)

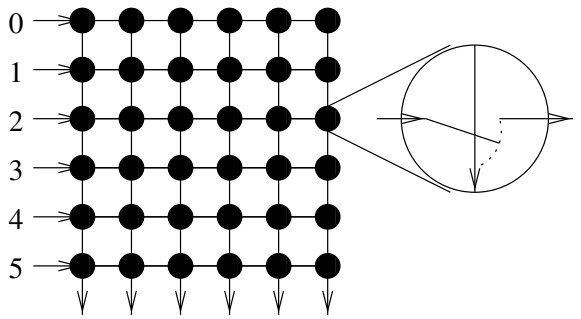
### 1.2.3 Weitere Varianten

- Synchrone Befehlsabarbeitung (SIMD)?
- Vektorrechner: Eine (oder mehrere synchron arbeitende) Arithmetik-Pipeline + **Leistungsfähiges Speichersystem**
- Hybride Ansätze. Zum Beispiel Netzwerk von kleinen Shared-Memory-Maschinen (SMP), Shared Memory Maschine aus Vektorrechnern Mesh of Multistage-Networks, ...



## 1.3 Verbindungsnetzwerke

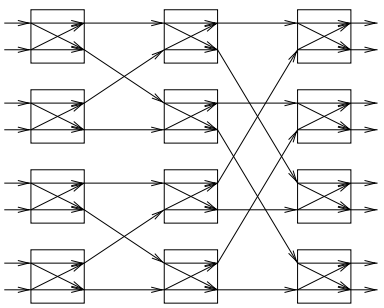
### 1.3.1 Kreusschienenverteiler




+ Schnell
+ Jede Permutation schaltbar
– $P^2$ Schalter nötig
- Beispiel (Sun E10000?)



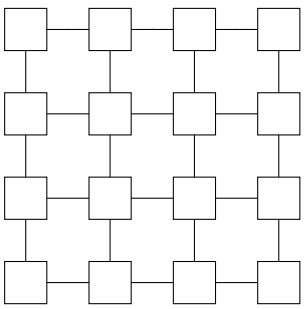
### 1.3.2 Mehrstufig



+ Nur $\Theta(\log P)$ Schalter pro Pfad
+/– Insgesamt $\Theta(P \log P)$ Schalter
+/– Nachrichten können sich in die Quere kommen
– Lange Draehte, $\Theta\left(P^2\right)$ VLSI-Fläche, $\Theta\left(P^{3/2}\right)$ Volumen
- Beispiel: IBM-SP, SB-PRAM, (Cluster am MPII?)



### 1.3.3 Gitter



+ Kosten linear in $P$
– Lange Wege
– Nachrichten behindern sich massiv. (nur $P^{1/2}$ können gleichzeitig einen Schnitt überqueren)
- Beispiel: Intel Paragon, ASCI red (9000 PEs!); Cray T3D/T3E (3-dim)

Varianten:

- 1-dimensional, 3-dimensional
- Ring, Torus, Diagonalverbindungen, ...

## 1.4 Analyse Paralleler Algorithmen

- Vergleich mit seq. Ausführungszeit $T_{\text{seq}}$
- Bestimme parallele Ausführungszeit $T_{\text{par}}$
- Speedup $S := \frac{T_{\text{seq}}}{T_{\text{par}}}$
  - Absolut: $T_{\text{seq}}$ ist „bestes“ sequentielles Programm
  - Relativ: $T_{\text{seq}}$ ist paralleles Programm für $P = 1$
- Effizienz $E := \frac{S}{P}$
  - Häufiges Ziel: $E$ =const
  - Sinnvolles Kostenmaß?
  - „Superlinearer“ Speedup ($E > 1$) möglich?

Einfaches Kostenmodell: Zeitbedarf für Nachricht der Länge $m$ ist $T_{\text{start}} + mT_{\text{byte}}$



## 1.5 Warum Nachrichtengekoppelte Programmierung

Ziele:

- Effizienz
- Portabilität
- Leichte Programmierbarkeit

Alternativen:

- Direkter Hardwarezugriff
- Parallele Programmiersprache
- Unterprogrammbibliothek



## 1.6 Unterprogrammbibliotheken für Parallelverarbeitung

- Herstellereigen: ggf. schnell aber nicht portabel
- PARMACS hat sich nicht durchgesetzt
- PVM: einfach, portabel, kostenlos, flexibel. Geringer Funktionsumfang, manchmal langsam.
- MPI (Message Passing Interface)
  - \+ Normiert
  - \+ Portabel
  - \+ Relativ Effizient
  - \+ Guter Kompromiss zwischen Einfachheit und hohem Funktionsumfang



## 1.7 Überblick über MPI

- Punkt-zu-Punkt-Kommunikation
- Kollektive Operationen. Beispiel: Broadcast
- (Datentypen)
- (Gruppen, Kontexte, Kommunikatoren)
- (Virtuelle Topologien)
- (MPI-2: Gemeinsamer Speicher, Prozesse, I/O, ... )
- (Was fehlt: Multithreading, ... )

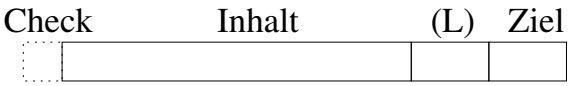
# 2 Routing

- Im Netzwerk
  - Wege auswählen
  - Pakete versus “Rohrleitung“
  - Pufferstrategien
- Ende-zu-Ende Protokolle
  - Pufferung
  - Synchronisation
  - Zuverlässigkeit
  - Reihenfolgetreue
  - Fairness
- Protokollstapel
- Beispiele



## 2.1 Router-(Hardware)

### 2.1.1 Netzwerkpakete

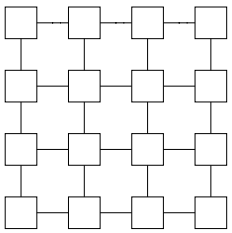



- Zielknoten (z.B. $x$, $y$ bei Gitter) $\neq$ Prozessor, Prozeß, ... Wichtig für dynamic routing / packet switching
- Länge (falls nicht fest)
- Inhalt (incl. Headerinformation höherer Schichten)
- ggf. Testbits für Fehlererkennung/Korrektur
- Bei shared memory:
  - Cache line fester Länge
  - Addresse



### 2.1.2 Wege Auswählen

$x$-$y$-Routing für Gitter.



1. Innerhalb einer Zeile zur richtigen Spalte
2. Innerhalb der Spalte

\+ Einfach, Schnell

– Kann zu unnötigen Stauungen führen

Fortgeschrittene Verfahren: adaptiv

\+ Höherer Durchsatz

– Kompliziert

– Reihenfolgetreue gefährdet



### 2.1.3 Store-and-Forward

- Ganzes Paket über die Leitung schicken bevor Weiterleitung beginnt

\+ Zuverlässig und gut analysierbar. Zum Beispiel ist $x$-$y$-Routing *im Mittel* annähernd optimal. $\leadsto$ effizienter randomisierter Algorithmus durch Umleitung über zufällige Zwischenstation.

– Hohe *Latenz* bei langen Paketen

– Benötigt viel Pufferspeicher

### 2.1.4 Cut-Through Routing

- Weiterleiten sobald nächstes Ziel erkannt
- \+ Sehr geringe Latenz bei niedriger Netzlast
- \+ Puffer sind nicht unbedingt notwendig. (*Wormhole-Routing* = Keine Puffer)
- – Nachrichten können sich leichter behindern
- – Schwer analysierbar
- Deadlockvermeidung wichtig
  - $x$-$y$-Routing bei Gittern
  - Zusätzliche Maßnahmen bei Tori („virtuelle Kanäle“)



## 2.2 Beispiele

### 2.2.1 IBM „High Performance Switch“

- Mehrstufiges Netzwerk aus $8 \times 8$-Kreuzschienenverteilern
- Umsetzung 8-bit *flits* 64-bit Pufferung
- Cut-Through-Routing mit begrenzter Pufferung
- 2.4$\mu$s Netzwerklatenz bei 512 PEs
- Mikrokanalkarte mit eigenem Kommunikationsprozessor



### 2.2.2 Dolphin SCI

- Wormhole-Routing auf unidirektionalen Ringen
- Switch kann mehrere Ringe über einen Bus koppeln (Store-and-Forward von Cache Lines)
- Adapterkarte für PCI Bus



### 2.2.3 Cray T3E

- 3D-Torus, 600MB/s Kanalbandbreite
- Cut-Through-Routing (Pufferung? Adaptiv?)
- Proprietäre Anbindung

## 2.3 Ende-zu-Ende-Protokolle

### 2.3.1 Schnittstelle zur Hardware

Typische Qualitätseigenschaften:

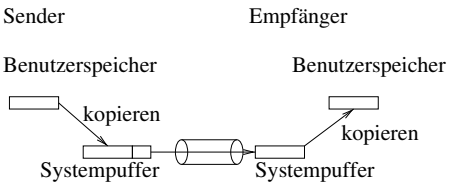
- Zuverlässig: Keine Paketverluste, Keine Verfälschungen ($\neq$ Telematik). **Wichtig für viele parallele Algorithmen**
- Reihenfolgetreu: Pakete von $A$ nach $B$ werden in der Absendereihenfolge ausgeliefert
- Fair: Kein Paket wird unbegrenzt lange von anderen Paketen aufgehalten.

Das heißt **nicht**, daß höhere Schichten dieselben Zusicherungen bekommen.

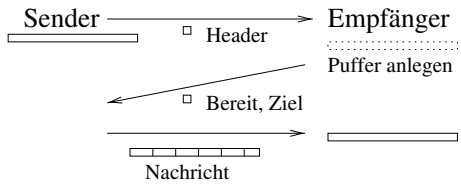


### 2.3.2 Pufferverwaltung

Kurze Nachrichten: Einwegeprotokoll




Lange Nachrichten: Dreiwegeprotokoll




+ Weniger Kopieroperationen
+ Universell
– Dreifache Anzahl Nachrichten
– Empfänger muss reagieren



### 2.3.3 Kanäle

Ein schnelles Zweiwegeprotokoll

- Genau ein Sender-Empfänger-Paar
- Vereinbarte Puffergröße
- Bestätigungen
- Sender muss nicht auf Bestätigungen warten wenn genug Puffer übrig (Sliding Window)
- Bei bidirektionelem Verkehr können Bestätigungen an Datenpakete gekoppelt werden (Piggibacking)

– Kann nicht direkt eingesetzt werden wenn unbekannt ist woher empfangen werden soll



## 2.4 Wer handhabt die Protokolle

- Hardware
  + Schnell
  + Anwendungsprogramme werden weniger aufgehalten
  – Unflexibel
  – Oft wenig Unterstützung
- Interrupt
  + Flexibel
  + Reagiert sofort
  – Muss mit Betriebssytem zusammenpassen
  – Langsam, vor allem bei heute üblichen Betriebssystemen und Prozessoren
- Polling
  + Einfach
  + Geringer Overhead
  – Unflexibel
  – Legt Prozessor lahm (Multithreading schwierig)

## 2.5 Beispiele

### 2.5.1 IBM SP

- Software auf Anwendungsprozessor (und Adapterkarte)
- 2 `memcpy` + 1 DMA + Adapteraktivität
- Eine Kopieroperation weniger für lange Nachrichten
- Polling (Interrupts als langsame Notlösung)
- ca. 40$\mu$s Latenz



### 2.5.2 Scali MPI

- (Uncached) Shared Memory wird benutzt um Kanäle zu implementieren.
- Kanäle werden direkter genutzt
- 1 `memcopy` + 1 DMA
- 3 Protokollvarianten für kurze, mittlere und lange Nachrichten
- Schnelleres Protokoll innerhalb eines SMP-Knotens
- Kurze Nachrichten werden direkt übertragen
- ca. 10$\mu$s Latenz



### 2.5.3 Cray T3E

- Hardwareunterstützung für kurze Nachrichten
- 2.7$\mu$s Ende-zu-Ende Latenz! (leider nicht für MPI)
- (nicht Cache kohärentes) Shared-Memory + Barrier in Hardware



## 2.6 Ein Schichtenmodell a la ISO/OSI

1. Physikalisch. Drähte, Glasfasern, Signalformen, Clocks
2. Verbindungsschicht (Data-Link-Layer). Kommunikation zwischen zwei Routern
3. Netzwerkschicht (Network-Layer). Verbindungsnetzwerk incl. Routing
4. Transportschicht. Fast der Service den MPI bietet. Nachrichten über das Netzwerk übertragen. (Zerstückeln, puffern, ... )
5. Sitzungsschicht. Leer?
6. Präsentationsschicht. Ggf. Konversion zwischen inkompatiblen Datenrepräsentationen
7. Anwendungsschicht. Weitergehende Protokolle, z.B. kollektive Operationen

# 3 Gehversuche mit MPI

```
#include <stdio.h>
#include "mpi.h"

int main(int argc, char** argv)
{
   MPI_Init(&argc, &argv);

   printf("Hello world!\n");

   MPI_Finalize();
}
```

- `g++ hello.c -o hello -lmpi`
- `mpirun -np 4 a.out`



## 3.1 Kommunikatoren

```
int main(int argc, char** argv)
{
   int myId, numProcs;

   MPI_Init(&argc, &argv);

   MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
   MPI_Comm_rank(MPI_COMM_WORLD, &myId);

   printf("I am PE %d out of %d PEs.\n",
          myId, numProcs);

   MPI_Finalize();
}
```

- Numerierte Gruppe von Prozessoren
- Kontext



## 3.2 Ping-Pong

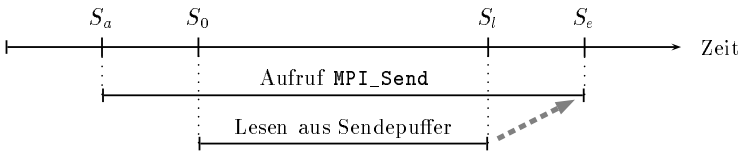
```
#define PING 42
#define PONG 17
if (myId == 0) {
   for (i = 0;  i < tries;  i++) {
      MPI_Send(buffer, length, MPI_DOUBLE,
               1, PING, MPI_COMM_WORLD);
      MPI_Recv(buffer, length, MPI_DOUBLE,
               1, PONG, MPI_COMM_WORLD,
               &status);
   }
} else {
   for (i = 0;  i < tries;  i++) {
      MPI_Recv(buffer, length, MPI_DOUBLE,
               0, PING, MPI_COMM_WORLD,
               &status);
      MPI_Send(buffer, length, MPI_DOUBLE,
               0, PONG, MPI_COMM_WORLD);
   }
}
```



## 3.3 MPI_Send

Die ersten drei Parameter sind

```
(void *buf,
int count,
MPI_Datatype datatype, .....)
```



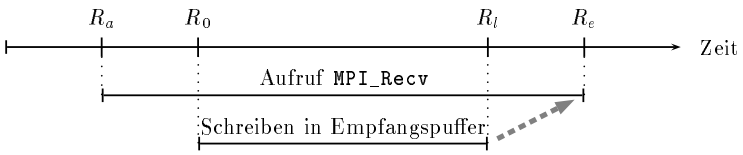

Die restlichen Parameter sind

```
(..... ,
int dest,
int tag,
MPI_Comm comm)
```

## 3.4 MPI_Recv

Die ersten drei Parameter sind

```
(void *buf,
int count,
MPI_Datatype datatype, .....)
```




Die restlichen Parameter sind `int source,`
`int tag,`
`MPI_Comm comm,`
`MPI_Status *status)`

- `MPI_ANY_TAG`
- `MPI_ANY_SOURCE`



**Ergebnis aus `status` ablesen**

**Absender:** `status.MPI_SOURCE`

**Tag:** `status.MPI_TAG`

**Länge:** `MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)` umständlich oder geschickt?

Zwei Operationen $S$ und $R$ passen zueinander:

- Gleicher Kommunikator `comm`
- $R$ hat von $S$ als `dest` angegebenen Rang
- $S$ hat von $R$ als `source` angegebenen Rang, oder $R$ hat `MPI_ANY_SOURCE` angegeben
- Gleiches `tag`, oder $R$ hat `MPI_ANY_TAG` angegeben
- Gleicher Datentyp ($\approx$)

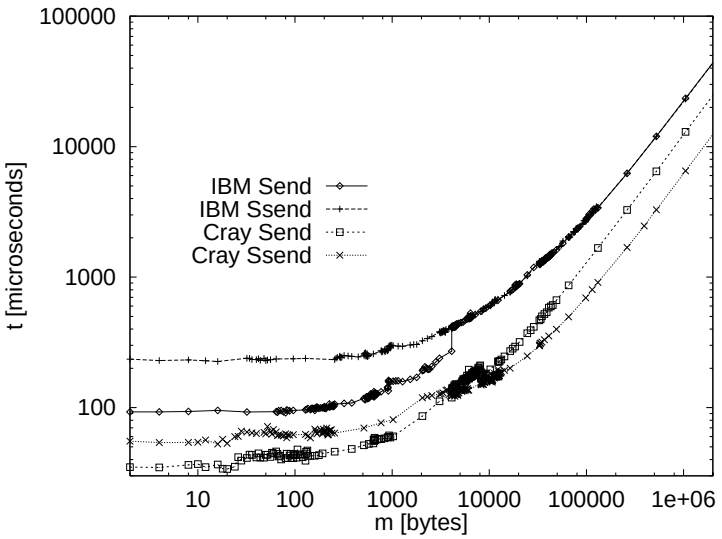


## 3.5 Zeitmessungen

- `double MPI_Wtime();` misst Zeit in Sekunden
- `MPI_Barrier(MPI_COMM_WORLD);` synchronisiert alle PEs. Wozu?
- Mittelwertbildung
- Ausreisser rausfiltern. (Wirklich?)



## 3.6 Ergebnisse auf IBM-SP und Cray T3E




Bandbreiten:

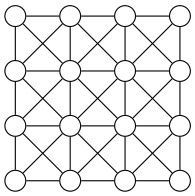
**IBM** $\approx$ 100 MB/s

**Cray** $\approx$ 300 MB/s

# 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

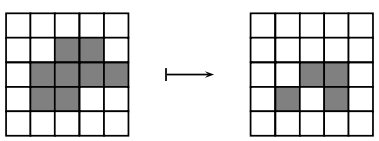
In einem Zeitschritt führen alle endlichen Automaten synchron einen Zustandsübergang aus.



### 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 |





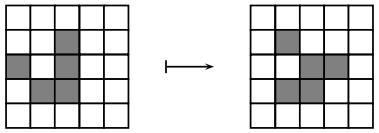
**Ein Modell für "Entmischung"**

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

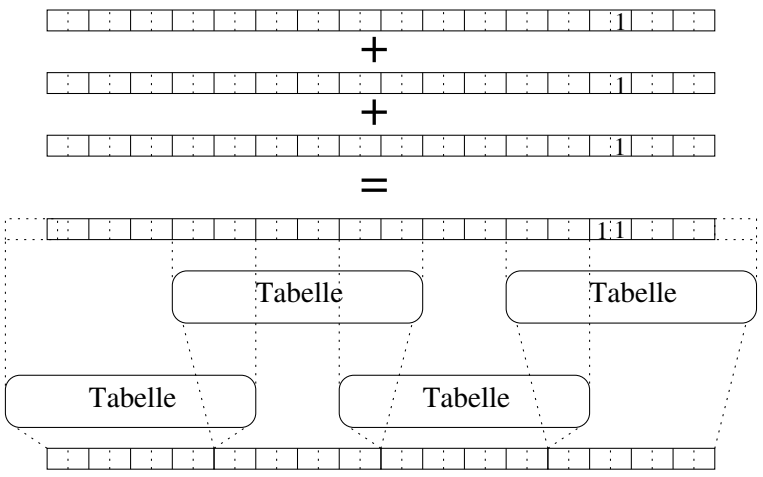
```
/*  treat boundary conditions
 *  IN:
 *  buf:       pointer to a stored configuration
 *  configRows:number of rows of configurations
 *  OUT:
 *  buf:       configuration is modified in plac
 */
static void boundary(Row *buf, int configRows)
{ int x,y;

  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];
  }
}
```



### 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 ($512 \times 512$ 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;
    }
  }
}
```



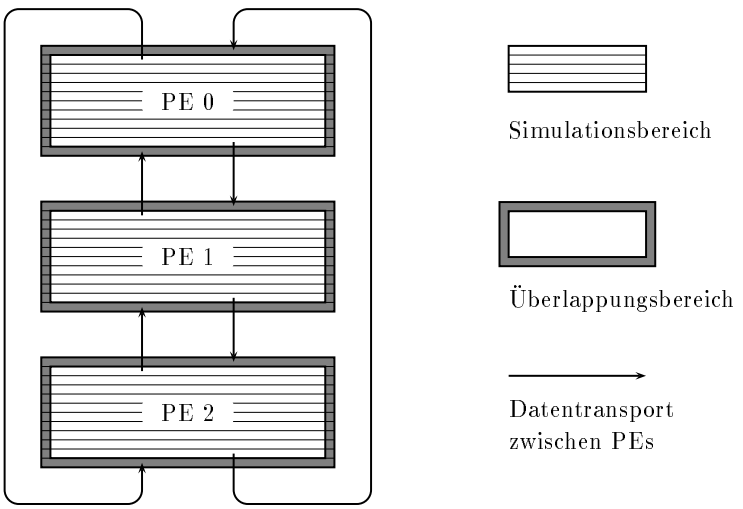
Weitere Maßnahmen:

- 64 Bit
- Paralleler Lookup für 8 Zellen. Wirklich?
- 1 Bit pro Zustand
  - − $\delta$ 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);
```

⇝

```
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 ser
  void *recvbuf, int recvcount, MPI_Datatype rec
  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

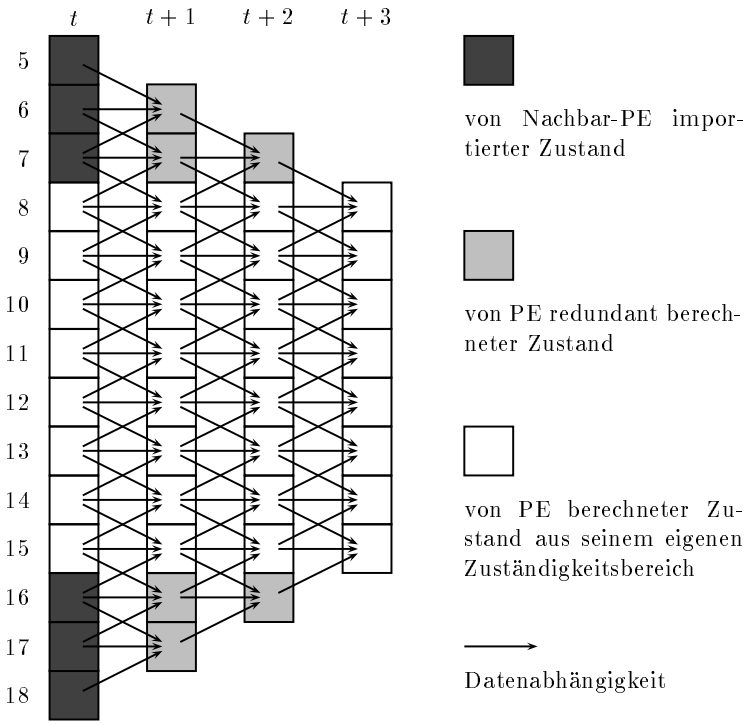


### Analyse

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



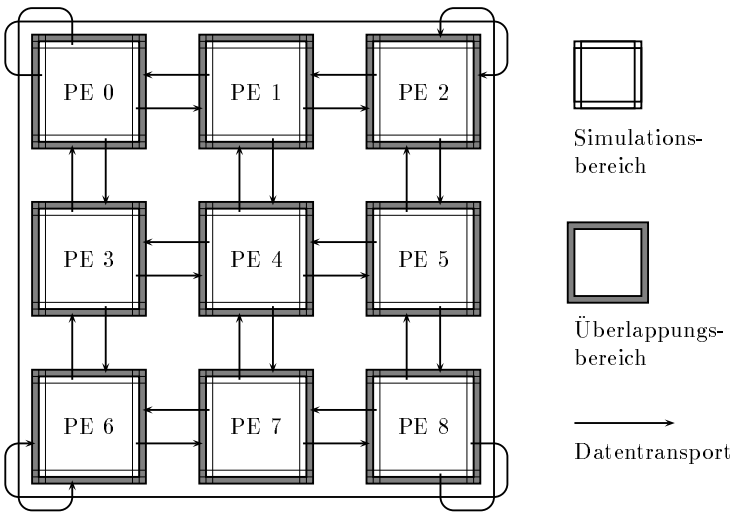



### 4.2.5 Analyse

- 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?



### 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:`

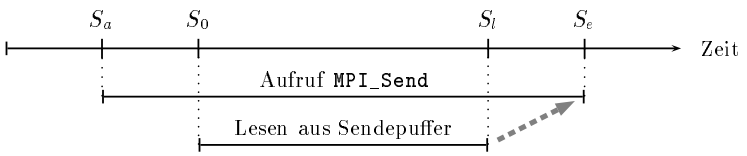
```
setenv PATH ${PATH}:/opt/mpi-1.0.13/bin
setenv PATH ${PATH}:/opt/mpi-1.0.13/lib/i86pc/ch_p4
setenv PATH ${PATH}:/KM/usr/sanders/praktikum/bin
```

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



# 5 Punkt-zu-Punkt-Kommunikation in MPI

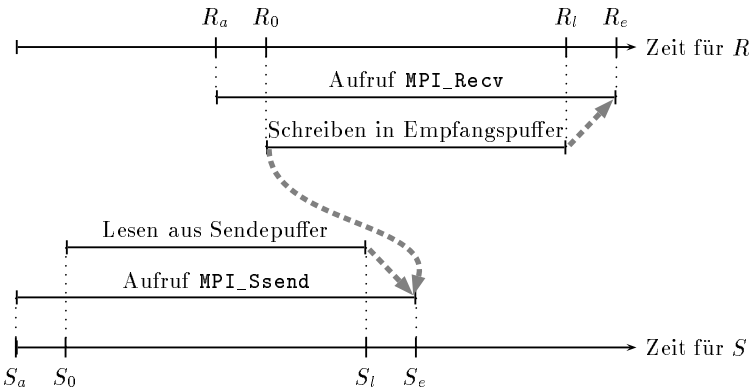
## 5.1 Blockierende Sendeoperationen




- `MPI_Send` kennen wir das wirklich schon?
- `MPI_Ssend` synchron
- `MPI_Bsend` „buffered“
- `MPI_Rsend` Empfänger ist „ready“
- `MPI_Sendrecv`
- `MPI_Sendrecv_replace`



### 5.1.1 Zyklische Datenverschiebung




Eine (bestenfalls) zweifelhafte Lösung:

```
MPI_S(s)end(sBuf, length, MPI_DOUBLE,
           iP+1+nP % nP, 7, MPI_COMM_WORLD);
MPI_Recv(rBuf, length, MPI_DOUBLE,
           iP-1+nP % nP, 7, MPI_COMM_WORLD, &st
```

$\Longrightarrow$ Verklemmung möglich (garantiert)



### 5.1.2 Handgestrickt: Eine komplizierte Lösung

```
if (iP & 1) {
  send();
  recv();
else {
  recv()
  send();
}
```

- $P$ gerade: geht so
- $P$ ungerade: langsam (Warum?)
- $P = 1$ Oops

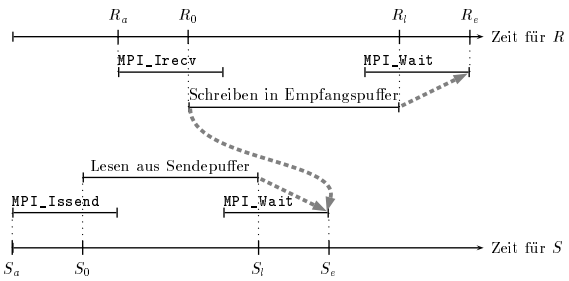
### 5.1.3 MPI_Bsend: Eine langsame Lösung

```
int size;
MPI_Pack_size(length, MPI_DOUBLE,
              MPI_COMM_WORLD, &size);
size += MPI_BSEND_OVERHEAD;
char *genug_speicher = malloc(size);
Assert(genug_speicher != 0);
MPI_Buffer_attach(genug_speicher, size);
...
MPI_Bsend(sBuf, length, MPI_DOUBLE,
          iP+1+nP % nP, 7, MPI_COMM_WORLD);
MPI_Recv(rBuf, length, MPI_DOUBLE,
          iP-1+nP % nP, 7, MPI_COMM_WORLD, &stat
...
int dummy;
char *dummyP;
MPI_Buffer_detach(&dummyP, &dummy);
```



## 5.2 Nichtblockierende Kommunikation

- Orthogonaler Satz von Sendeoperationen: `MPI_Isend`, `MPI_Issend`, `MPI_Ibsend` (sinnvoll?), `MPI_Irsend`
- Nichtblockierendes Empfangen: `MPI_Irecv`
- send bzw. receive stößt die Kommunikation an und bekommt ein `MPI_Request` zurück.
- Warten auf Komplettierung mit `MPI_Wait`. (u.v.a.m.)






### 5.2.1 Beispiel: Selbstbau von Sendrecv

```
void my_Sendrecv(void *sb, int sc, MPI_Datatype st,
                 int dst, int stag,
                 void *rb, int rc, MPI_Datatype rt,
                 int src, int rtag,
                 MPI_Comm comm, MPI_Status *status)
{ MPI_Status localStatus[2];
  MPI_Request req[2];

  MPI_Isend(sb, sc, st, dst, stag, comm, req + 0);
  MPI_Irecv(rb, rc, rt, src, rtag, comm, req + 1);
  MPI_Waitall(2, req, localStatus);
  *status = localStatus[1];
}
```



## 5.3 Weitere nichblockierende Funktionen

- `MPI_Waitany`, `MPI_Waitsome` fair!
- `MPI_Test`, `MPI_Testall`, `MPI_Testany`, `MPI_Testsome`
- Persistente requests: `MPI_Send_init`, ... `MPI_Start`

### 5.3.1 Nichtblockierende Kommunikation Sinnvoll?

– Komplizierter

– U.u. größerer overhead (mehr Funktionsaufrufe, ... )

\+ Deadlockvermeidung

\+ u.U. Höhere Bandbreite durch bidirektionale (+multiported?) Kommunikation

• Überlappung von Kommunikation und Berechnung (Kommunikationszeitverdeckung, latency hiding) Aber: U.u. niedrigere Bandbreite bei bestimmten Implementierungen (IBM SP?)



### 5.3.2 Kommunikationszeitverdeckung zur Zellularautomatensimulation

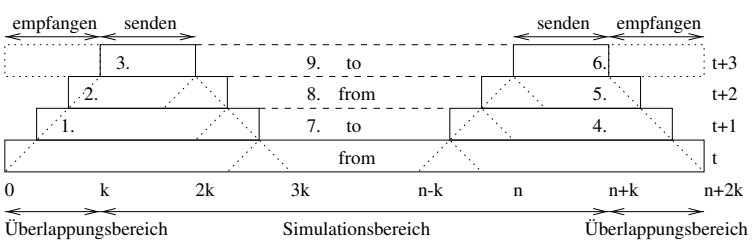
Hauptschleife:

- Simuliere Zeilen 1 und $n$
- Starte Kommunikation für die nächste Iteration
- Simuliere Zeilen 2 bis $n-1$
- Warte auf Komplettierung der der Kommunikation



### 5.3.3 Entkopplung + Überlappung

**PROCEDURE** simulate($x$, $y$, $k$) (* $k$ odd *)
**for** $i := k-1$ **downto** 0 **do**
  simulate lines $k-i$ through $2k+i$, $x \to y$
  simulate lines $n-i$ through $n+k+i$, $x \to y$
  swap $x$ and $y$
start send lines $x[k]$ through $x[2k-1]$ up
start send lines $x[n]$ through $x[n+k-1]$ down
start receive lines $y[0]$ through $y[k-1]$ from above
start receive lines $y[n+k]$ through $y[n+2k-1]$ from belo
**for** $i := 0$ **to** $k-1$ **do**
  simulate lines $3k-i-1$ through $n-k+i$, $y \to x$
  swap $x$ and $y$
complete communication






## Blick über den Tellerrand

- Anwendungen
  - Stömungssimulation, Spin-Gläser, Bildverarbeitung
  - Explizite iterative Lösung diskretisierter partieller Differentialgleichungen
  - Zeitabhängige partielle Differentialgleichungen
  - Partikelsimulation
- Weitere Algorithmische Probleme
  - Lastverteilung
  - Komplizierte Randbedingungen
  - Unregelmäßige Gitter, Schrittweiten und Überführungsfunktionen
  - Variable Gitter
  - Pyramiden bzw. Multigrid

# 6 Kollektive Operationen

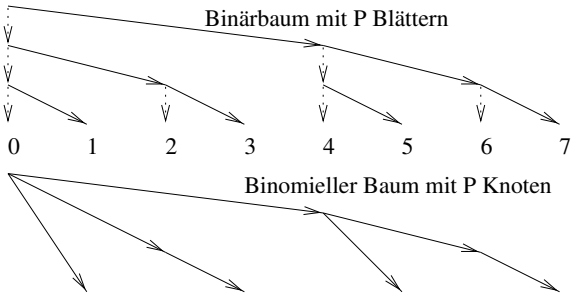
Wir kennen bereits:

- `MPI_Barrier`
- `MPI_Gather`



## 6.1 Broadcast: Implementierung

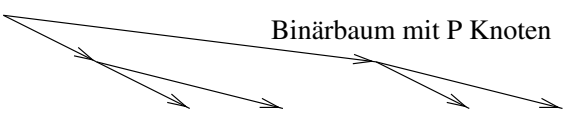
- root=0
- $P = 2^k$ am einfachsten
- kurze Nachrichten
- $T_{\text{broadcast}}(P, n) \approx (T_{\text{start}} + nT_{\text{byte}}) \log P$
- $P - 1$ Nachrichten a $n$ Byte
- Genaueres Modell (LogP ... ) führt zu leicht modifiziertem Kommunikationsmuster. (ausprobieren?)
- Multicast?






### 6.1.1 Pipelined Broadcast

- $P = 2^k - 1$
- **lange** Nachrichten
- $T_{\text{broadcast}}(P, n) \approx 3nT_{\text{byte}} + AT_{\text{start}} \log P$, $A \approx 3$–5?
- $P - 1$ Nachrichten a $n$ Byte
- Genaueres Modell (LogP, ... ) führt zu leicht modifiziertem Kommunikationsmuster. (ausprobieren?)
- **extrem lange** Nachrichten?

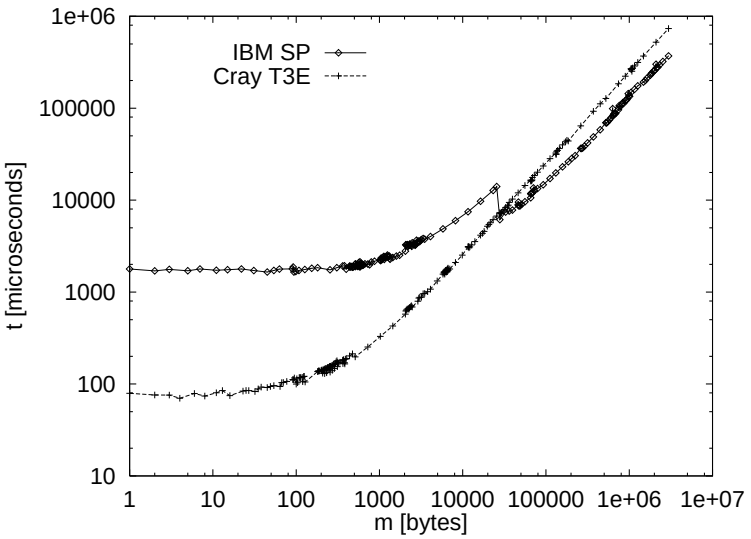





## 6.2 Reduktion

- $\otimes$ sei assoziativer (optional zus. kommutativer) Operator.
- Berechne $\bigotimes_{i<P} x_i$, wobei $x_i$ ein auf PE $i$ gelegener Vektor der Länge $n$ ist.
- Pipelined tree etc.: $T_{\text{reduce}} = \text{O}(nT_{\text{byte}} + T_{\text{start}} \log P)$
- Viele vordefinierte Operatoren. Genug?

### 6.2.1 IBM SP versus Cray T3E






# 7 Lineare Algebra

- Allgegenwärtig im wissenschaftlichen Rechnen
- Prozessoren sind darauf optimiert
- Andere Algebren (z.B. boolsch) $\leadsto$ Graphenalgorithmen, ...



## 7.1 Matrizenrechnung

- $\mathbf{A} \in \mathbf{R}^{r \times s}$, $\mathbf{B} \in \mathbf{R}^{s \times t}$ Matrizen mit $\mathbf{A} = (a_{ij})$ und $\mathbf{B} = (b_{ij})$
- $\mathbf{R}$ Ring (später: Körper)

$$c_{ij} = \sum_{k=1}^{s} a_{ik} \cdot b_{kj}$$



### 7.1.1 Geblockte Schreibweise

$$\mathbf{A} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \text{ und } \mathbf{B} = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

$$\mathbf{C} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

$$= \begin{pmatrix} A_{11} \cdot B_{11} + A_{12} \cdot B_{21} & A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\ A_{21} \cdot B_{11} + A_{22} \cdot B_{21} & A_{21} \cdot B_{12} + A_{22} \cdot B_{22} \end{pmatrix}$$

Allgemein:

$$C_{ij} = \sum_{k} A_{ik} \cdot B_{kj}$$

(nicht notwendig quadratisch)

**Anwendungen der Zerlegung**

- Besserer sequentieller Code (warum?)
- Asymptotisch schnellere Algorithmen (Strassen $O(n^{2.81})$, Winograd ... )
- Parallelisierung



**Bibliotheksfunktionen**

Basic Linear Algebra Subroutines (BLAS)

- Meist **viel** effizienter als direkte Implementierung.
  - Blockung an Speicherhierarchie anpassen
  - Alinierung
  - Prefetching
  - Schleifen ausrollen
  - Software-Pipelining
  - Register nutzen
  - Instruction-Scheduling
- Aufgabe 5.2. Eigenes Programm versus Bibliothek `praktikum/ATLAS/atlas.ps,README` Architektur `PII` Compiler `pgcc -O6` oder `pcc -fast -O4`.
- Weitere Info `http://www.netlib.org/atlas,blas`
- Auch Parallel



### 7.1.2 Matrixmultiplikation: Ein trivialer Algorithmus

**„Unfaire“ Ein- Ausgabebedingungen**

- Eingabe: Alle PEs besitzen Kopien beider Matrizen
- Berechnung/Ausgabe: PE $i$ berechnet Spalten $in/P \leq j < (i+1)n/P$

**Realistischer: PE 0 macht Eingabe und Ausgabe**

1. Broadcast **A**
2. Scatter Spalten von **B** (MPI-Datentypen!)
3. PE $i$ berechnet Spalten $in/P \leq j < (i+1)n/P$
4. Gather **C**



### 7.1.3 Analyse

- $n \times n$ Matrizen
- $P$ teilt $n$
- $\approx 2n^3/PT_{\text{flop}}$ für arithm. Operationen
- $T_{\text{gather}} = T_{\text{scatter}} \geq 8n^2 T_{\text{byte}}$
- $T_{\text{broadcast}} =\geq 2 \cdot 8n^2 T_{\text{byte}}$
- $T_{\text{par}} \geq 2n^3/PT_{\text{flop}} + 32n^2 T_{\text{byte}}$
- Gute Effzienz nur wenn

$$2n^3/PT_{\text{flop}} \gg 32n^2 T_{\text{byte}}$$
$$\Leftrightarrow \quad n \gg 16P\frac{T_{\text{byte}}}{T_{\text{flop}}}$$

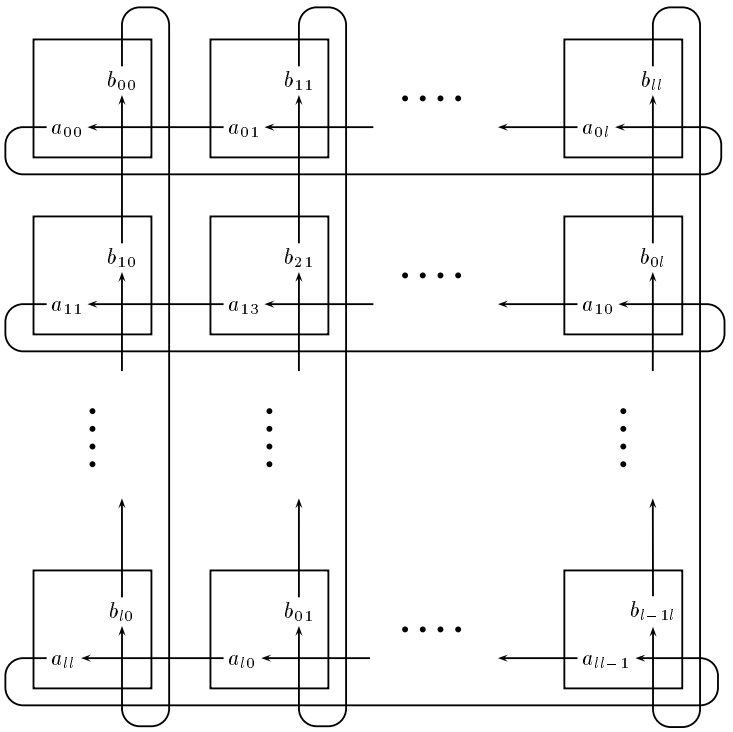
Beispiel $T_{\text{flop}} = 4\text{ns}$, $T_{\text{byte}} =300\text{ns}$, $P = 16$:

$$n \gg 19200$$

Speicherbedarf: mehr als 8GB!

### 7.1.4 Matrixmultiplikation nach Cannon

$$c_{ij} \leftarrow c_{ij} + a_{i,(i+j+k) \bmod n} \cdot b_{(i+j+k) \bmod n,j}$$






**Analyse**

- $P$ Quadratzahl
- Blockgrößen $n/\sqrt{P} \times n/\sqrt{P}$
- $\sqrt{P}$ Schritte
- $\approx 2(n/\sqrt{P})^3 T_{\text{flop}}$ für arithm. Operationen
- Kommunikation a $4(T_{\text{start}} + (8n/\sqrt{P})^2 T_{\text{byte}})$
- $T_{\text{par}} = 2n^3/PT_{\text{flop}} + 4\sqrt{P}T_{\text{start}} + 32n^2 T_{\text{byte}}/\sqrt{P})$
- Gute Effzienz falls
  - $2n^3/PT_{\text{flop}} \gg 4\sqrt{P}T_{\text{start}}$
  - $2n^3/PT_{\text{flop}} \gg 32n^2 T_{\text{byte}}/\sqrt{P})$
  - $n \gg \sqrt{P}\max(\sqrt[3]{2T_{\text{start}}/Tflop}, 16T_{\text{byte}}/T_{\text{flop}})$

Beispiel $T_{\text{flop}} = 4$ns, $T_{\text{start}}$ =300 000ns,
$T_{\text{byte}}$ =300ns, $P = 16$:

$$n \gg \max(212.5, 4800)$$

Also ist Bandbreite das Hauptproblem.
Speicherbedarf bei $n = 4800$:
$3 \cdot 8 \cdot 4800^2 \approx 550MB$



**Systolische Algorithmen allgemein**

- Immer die gleichen Befehle (im wesentlichen)
- Wandernde Daten
- Synchrone Arbeitsweise
- Vorzugsweise einfache Kommunikationsmuster
- Gut für Hardwareentwurf



**Verfeinerungen**

- MPI Prozesstopologien
  - `MPI_Cart_create` numeriert die PEs so um daß tatsächlich Nachbarschaftskommunikation stattfindet
  - `MPI_Dims_create` hilft geeignete Dimensionen finden
  - Diverse andere Hilfsfunktionen. Koordinatenumrechung etc. `MPI_Cart_rank`, `MPI_Cart_shift`, ...

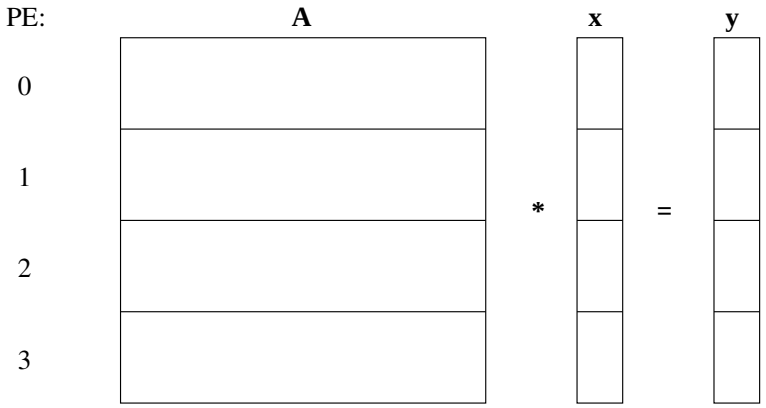
## 7.2 Matrix-Vektormultiplikation

- Degenerierte Matrix-Matrixmultiplikation ...
- Basis für iterative Gleichungslöser, Eigenwertprobleme, etc.

$$T_{\text{seq}} = 2n^2 T_{\text{flop}} + \mathrm{O}(n)$$



### Zeilenweise Zerlegung






### Analyse

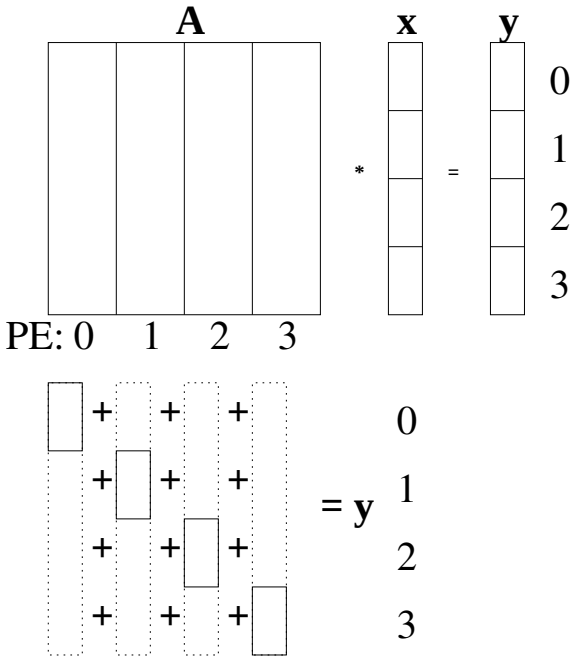
$$\begin{aligned} T_{\text{par}} &\approx 2\frac{n^2}{P} T_{\text{flop}} + T_{\text{all-gather}}(n/P, P) \\ &\approx 2\frac{n^2}{P} T_{\text{flop}} + 2 \cdot 8n T_{\text{byte}} \end{aligned}$$

Gute Effizienz für

$$n \gg 8nP \frac{T_{\text{byte}}}{T_{\text{flop}}}$$



### Spaltenweise Zerlegung



**Analyse**

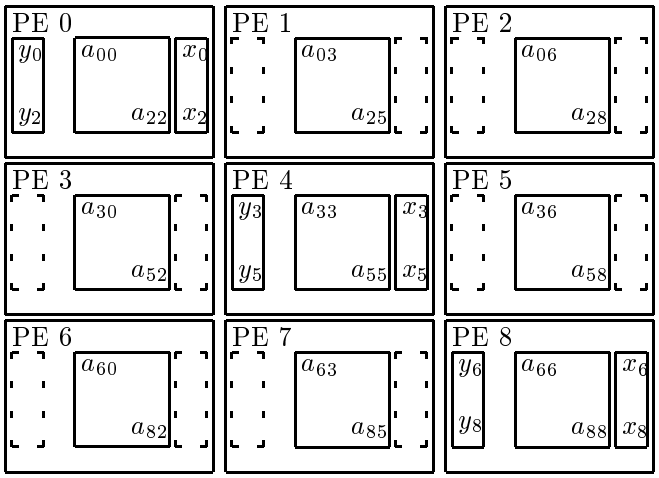
- Jeder kriegt ganzes Ergebnis:
  $T_{\text{par}} \approx 2\frac{n^2}{P}T_{\text{flop}} + T_{\text{allreduce}}(n, P)$
- $P$ Teilsummationen:
  $T_{\text{par}} \approx 2\frac{n^2}{P}T_{\text{flop}} + PT_{\text{reduce}}(n/P, P)$
- Jeder sammelt seinen Teil und summiert dann:
  $T_{\text{par}} \approx 2\frac{n^2}{P}T_{\text{flop}} + T_{\text{all-to-all}}(n/P, P)$
  Nur bei vollstaändiger Verknüpfung konkurrenzfähig.
- Die maßgeschneiderte MPI Funktion:
  $T_{\text{par}} \approx 2\frac{n^2}{P}T_{\text{flop}} + T_{\text{reduce-scatter}}(n/P, P)$

Gute Effizienz für

$$n \gg 16nP\frac{T_{\text{byte}}}{T_{\text{flop}}}$$



### 7.2.1 Zweidimensionale Zerlegung






### 7.2.2 Analyse

$$\begin{aligned} T_{\text{par}} &\approx 2\frac{n^2}{P}T_{\text{flop}} + T_{\text{broadcast}}(n/\sqrt{P}, \sqrt{P}) + \\ &\quad T_{\text{reduce}}(n/\sqrt{P}, \sqrt{P}) \\ &\approx 2\frac{n^2}{P}T_{\text{flop}} + 4 \cdot 8nT_{\text{byte}}/\sqrt{P} \end{aligned}$$

Gute Effizienz für $n \gg 16\frac{T_{\text{byte}}}{T_{\text{flop}}}\sqrt{P}$.



### 7.2.3 `MPI_Comm_split`

```
int MPI_Comm_split(
  MPI_Comm comm, int color, int key,
  MPI_Comm *newComm);
```

- Kollektiv auf `comm`
- Jede Farbe gibt einen Kommunikator (zurückgegeben in `newComm`)
- $\texttt{key}_i < \texttt{key}_j \implies \texttt{rank}_i < \texttt{rank}_j$
- `MPI_UNDEFINED`

Beispiel:

```
MPI_Comm_size(MPI_COMM_WORLD, &P);
MPI_Comm_rank(MPI_COMM_WORLD, &i);
MPI_Comm_split(MPI_COMM_WORLD, i >= P/2, 0,
               &myHalf);
```

### 7.2.4 Der DNS-Algorithmus zur Matrixmultiplikation

- Dekel, Nassimi und Sahni
- KumEtAl94 Abschnitt 5.4.4.
- Blockgröße $n/\sqrt[3]{P} \times n/\sqrt[3]{P}$
- $T_{\text{par}} \approx$
  $2n^3/PT_{\text{flop}} + 2T_{\text{broadcast}}(n^2/P^{2/3}, P^{1/3}) +$
  $T_{\text{reduce}}(n^2/P^{2/3}, P^{1/3})$
- Effiziente Broadcast und Reduce-Operationen wichtig

\+ Kommunikationsvolumen nur $\mathrm{O}\left(n^2/P^{2/3}\right)$ pro PE

– Speicherbedarf insgesamt $\Omega\left(n^2 P^{1/3}\right)$

– Mindestens $3D$-Gitter nötig (zumindest asymptotisch)



## 7.3 Lösen Linearer Gleichungssysteme

- Viele Verfahren
- Abhängig von
  - Struktur der Matrix
  - Erwägungen bzgl. Numerische Stabilität
- Hier: Gausselimination mit Zeilen-Pivot-Suche
- Ein *direktes* Verfahren
- Verwandte Probleme:
  - Gauss-Jordan-Elimination
  - LU Zerlegung
  - Matrixinversion
  - Cholesky Zerlegung

Hier: Gausselimination mit Zeilen-Pivot-Suche



### 7.3.1 Mathematische Formulierung

$n \times n$ Matrix

$$\mathbf{A} = (a_{ij})$$

$(0 \leq i, j < n)$ löse

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$$



### 7.3.2 Algorithmus in Vektorschreibweise

**PROCEDURE** GaussElimination($\mathbf{A}' = [\mathbf{A}|b]$) : $n$-Vector
**for** $k := 0$ **to** $n-1$ **do**

(* Pivoting *) find $k \leq i < n$ such that $a_{ik}$ is maximi
exchange rows $k$ and $i$

(* make a 1 *)
multiply row $k$ by $1/a_{kk}$

(* Zero out column below *)
$\mathbf{a}_{j*} = \mathbf{a}_{j*} - a_{jk}/a_{kk}\mathbf{a}_{k*}$ for $k < j < n$

(* Back substutution *)
**return** backSubstitute($\mathbf{A}'$)

$$\begin{aligned} T_{\text{seq}} &= \sum_k 2(n-k-1)^2 T_{\text{flop}} + \mathrm{O}(n-k) \\ &= \frac{2}{3} n^3 T_{\text{flop}} + \mathrm{O}\left(n^2\right) \end{aligned}$$

### 7.3.3 Spaltenweise Blockverteilung

- PE $l$ is für Spalten mit $l(n+1)/P \leq i < (l+1)(n+1)/P$ zuständig.
- Broadcast von Spalte $j$ ($a_{kj}$ für $k \leq j < n$)
- Kommunkationskosten $\sum_k T_{\text{broadcast}}(n-k+1, P-\frac{kP}{n})$ $\approx 8nT_{\text{byte}}$
- Lokale Berechnung: $\sum_k 2(n-k-1)\frac{n}{P}T_{\text{flop}}$ $= \frac{n^3}{P}T_{\text{flop}} + \mathrm{O}\left(\frac{n^2}{P}\right)$. Oops.
- Problem: Eine zunehmende Zahl von Prozessoren wird arbeitslos.



### 7.3.4 Spaltenweise zyklische Verteilung

PE $l$ ist für Spalten mit $i \bmod P = l$ zuständig.

$$T_{\text{par}} = \frac{2}{3}n^3 T_{\text{flop}} + \mathrm{O}\big(n^2 P T_{\text{flop}} + nT_{\text{broadcast}}(n,P)\big)$$



**Entkopplung**

- Spaltenweise Blockverteilung:
  - Fasse $m$ Iterationen zusammen
  - Bringe $m \times m$ Matrix in Dreiecksform $\rightsquigarrow 2/3m^3 T_{\text{flop}}$ Wartezeit
  - Ein Broadcast $m$ facher Länge statt $m$ einfache Broadcasts $\rightsquigarrow \mathrm{O}(f(P)m)$ gesparte Startups ($f(P)$ zwischen $\log p$ und $P$).
  - Optimales $m$ in $\sqrt{\frac{T_{\text{start}} f(P)}{2T_{\text{flop}}}}$
- So nicht bei zyklischer Verteilung
- Lösung block-zyklische Verteilung mit Blöcken der Größe $m$ Bedingung: $mP \ll n$.



# 8 Lineare Algebra für dünnbesetzte Matrizen

## 8.1 Tridiagonale Gleichungssysteme

T. Leighton, *Introduction to Parallel Algorithms and Architectures*, Morgan Kaufmann, 1992. Abschnitt 1.3.3.

W. H. Press, S.A. Teukolsky, W. T. Vetterling, B. P. Flannery, *Numerical Recipes in C*, Cambridge University Press, 2. Ausgabe, 1992. Abschnitt 2.4.

$$A := \begin{pmatrix} d_1 & u_1 & & & \\ l_2 & d_2 & u_2 & & \\ & \ddots & \ddots & \ddots & \\ & & l_{n-1} & d_{n-1} & u_{n-1} \\ & & & l_n & d_n \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix}$$

$A$ sei symmetrisch und positive definit ($\forall v : v \cdot A \cdot v > 0$) oder diagonaldominant ($|d_j| > |l_j| + |u_j|$).

### 8.1.1 Gausselimination

- Speicher $5n$ Zahlen
- $n$ Divisionen (extra zählen)
- $3n$ Subtraktionen, $4n$ Multiplikationen
- Kein Pivoting nötig
- Inhärent sequentiell



### 8.1.2 Odd-Even-Reduktion

- Substituiere
$$x_i = \frac{1}{d_i}(b_i - l_i x_{i-1} - u_i x_{i+1})$$
für ungerade $i$ (mit $x_0 = x_{n+1} = 0$)
- Setze in Gleichungen für gerade $i$ ein
- Löse rekursiv das System der Größe $\lfloor n/2 \rfloor$
- Rücksubstitution
- Deutlich mehr arithmetische Operationen als Gausselimination ($\approx 17n$ Multiplikationen)



## Lösung über eine lineare Rekurrenz

$$\begin{pmatrix} x_{i+1} \\ x_i \\ 1 \end{pmatrix} = \underbrace{\begin{pmatrix} -\frac{d_i}{u_i} & -\frac{l_i}{u_i} & \frac{b_i}{u_i} \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}}_{G_i} \begin{pmatrix} x_i \\ x_{i-1} \\ 1 \end{pmatrix}$$

also

$$\begin{pmatrix} x_{i+1} \\ x_i \\ 1 \end{pmatrix} = \prod_{j=i}^{1} G_i \begin{pmatrix} x_1 \\ 0 \\ 1 \end{pmatrix}$$

- Berechnung mittels paralleler Postfixberechnung (wie auf Präfix umleiten?)
- Beispiel für nichtassoziative Operation
- Oft numerisch instabil
- Ebenfalls viele arithm. Operationen
- Lösung durch 3 aufeinanderfolgende Rekurrenzberechnungen für $2 \times 2$ Matrizen ist besser. (Parallele LU-Zerlegung)

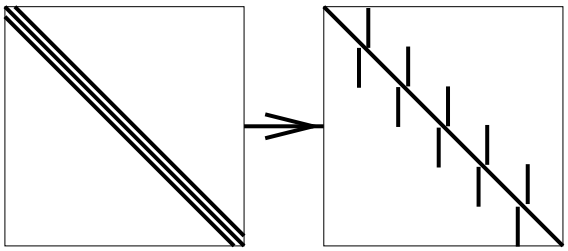


### MPI: Präfixsummen

`MPI_Scan` ähnelt `MPI_Reduce`. Nur wird nicht ein einziges Ergebnis produziert, sondern auf PE $i$ wird $a_0 \otimes a_1 \otimes \cdots \otimes a_i$ bestimmt. Der Prototyp ist

```
int MPI_Scan(void *sendbuf,
             void *recvbuf, int count,
             MPI_Datatype datatype,
             MPI_Op op,
             int root, MPI_Comm comm);
```

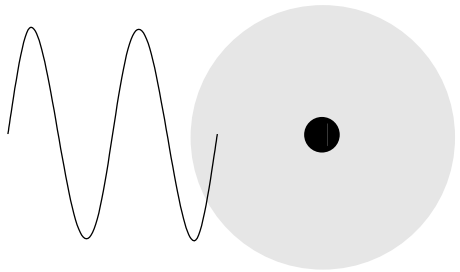
### 8.1.3 Lokale Problemreduktion



- $n \times n \rightarrow 2P \times 2P$
- Etwas effizienter für große $n$ ($\approx 17n$ Flops)



## 8.2 Anwendungsbeispiel: 2D zeitabhängige Dirac-Gleichung



- $10^{19}$ W/cm$^2$, Elektron erreicht relativistische Geschwindigkeiten
- 2D um elektrisches und magnetisches Feld zu erfassen
- Hohe Auflösung um de Broglie Wellenlänge des Elektrons zu erfassen
- $\rightarrow$ kleine Zeitschritte
- Viele Zeitschritte um "niedrige" Schwingungsfrequenz des Lichts zu erfassen



### 8.2.1 Die partielle Differentialgleichung

$$\mathrm{i}\frac{\partial}{\partial t}\Psi(x,z;t) =$$
$$\left\{c\alpha\cdot\left[\mathbf{p}-\frac{1}{c}\mathbf{A}(x,z;t)\right]+c^2\beta+U(x,z)\right\}\Psi(x,z;t)$$



### 8.2.2 Numerischer Ansatz

- Räumliche und zeitliche Diskretisierung
- Differentialoperatoren $\rightarrow$ Differenzenoperatoren
- Alternating direction implicit method
- Koeffizienten sind $4 \times 4$ komplexe Matrizen mit spezieller Struktur
- Problem in $z$-Richtung hat identische a priori bekannte Koeffizienten $\rightarrow$ weniger Arithmetik

### 8.2.3 Die Differenzengleichung

$$\frac{1}{\Delta t}[\Psi^{t+1/2}(j,k) - \Psi^{t}(j,k)] =$$
$$-\frac{c}{2}(\alpha_x^{(u)}\Delta_j^{+} + \alpha_x^{(l)}\Delta_j^{-})\Psi^{t+1/2}(j,k)$$
$$-\frac{c}{2}(\alpha_z^{(u)}\Delta_k^{+} + \alpha_z^{(l)}\Delta_k^{-})\Psi^{t}(j,k) +$$
$$+\frac{i}{2}[\alpha_z A^{t}(j,k) - \beta c^2 + V(j,k)]\Psi^{t}(j,k),$$
$$\frac{1}{\Delta t}[\Psi^{t+1}(j,k) - \Psi^{t+1/2}(j,k)] =$$
$$-\frac{c}{2}(\alpha_x^{(u)}\Delta_j^{+} + \alpha_x^{(l)}\Delta_j^{-})\Psi^{t+1/2}(j,k)$$
$$-\frac{c}{2}(\alpha_z^{(u)}\Delta_k^{+} + \alpha_z^{(l)}\Delta_k^{-})\Psi^{t+1}(j,k) +$$
$$+\frac{i}{2}[\alpha_z A^{t+1}(j,k) - \beta c^2 + V(j,k)]\Psi^{t+1}(j,k)$$



**Zeilenweise Zerlegung**

+ Schwieriger Schritt lokal
- Leichter Schritt: $n$ unabhängige block-tridiagonale LGS

**Transposition**

+ Alles lokal $\rightarrow$ wenig Arithmetik
– Seeehr viel Kommunikation

**2D-Zerlegung**

+ Weniger Kommunikation
– Alle Systeme müssen durch verteilten Alg. gelöst werden



### 8.2.4 $m$ unabhängige Systeme

**All-to-all Kommunikation:**

- PE $i$ sammelt die reduzierten Systeme $in/P \leq j < (i+1)n/P$
- $T_{\text{comm}} = T_{\text{all-to-all}}(\mathrm{O}(n/P), P)$

**Parallele Lösung der Systeme**

- In der Literatur beschriebene Verfahren $T_{\text{comm}} = \mathrm{O}((T_{\text{start}} + nT_{\text{byte}}) \log P)$
- Mit Pipelining $T_{\text{comm}} = \mathrm{O}(T_{\text{start}} \log P + nT_{\text{byte}})$

– Kompliziert



**Sequentielle Lösung der Einzelsysteme**

+ Optimale Anzalh arithm. Operationen
+ Ergebnis auf jeden Fall unabh. von Prozessorzahl
- Chunks von $k$ Systemen auf einmal
  – Startups bis Ergebnisse eintrudeln: $4PT_{\text{start}} \ll n^2 T_{\text{coeff}}/P$, d.h. $n \gg 2PT_{\text{start}}/T_{\text{coeff}}$
  – Berechnungen bis Ergebnisse eintrudeln $knT_{\text{coeff}} \ll n^2 T_{\text{coeff}}/P$, d.h. $k \ll n/P$
  – Gesamtzahl Startups $4nT_{\text{start}}/k \ll n^2 T_{\text{coeff}}/P$, d.h. $k \gg \frac{PT_{\text{start}}}{nT_{\text{coeff}}}$

**Verallgemeinerungen**

- Diagonalen bestehen selbst aus Matrizen (blocktridiagonale Systeme) (arithm. Aufwand?)
- $k$ Diagonalen (arithm. Aufwand?)
- Bandstruktur muss erst gefunden werden (i.allg. NP vollst)
- $d$-D ADI
- Matrizen ohne Bandstruktur: Problem ist Vermeidung von **fill in**



## 8.3 Matrix-Vektor-Multiplikation

- Anzahl nicht-null Elemente oft in $O(n)$
- Repräsentation der Matrix
- Grundprinzip der Parallelisierung: Jedes PE bekommt Teilmenge der Zeilen
- `MPI_Allgather` des Vektors oft zu teuer
- Idee: Vektor nur dahin schicken wo gebraucht
- $\rightarrow$ Lokalitätserhaltende Verteilung wichtig



## 8.4 Iterative Gleichungslöser

### 8.4.1 Gauss-Seidel Iteration

Idee: Wenn $x(k)$ eine gute Näherung ist, dann ist

$$x_i(k+1) := \frac{1}{a_{ii}}(b_i - \sum_{j \neq i} a_{ij} x j)$$

eine Verbesserung.

- Paralleler Update für *unabhängige* Indizes aus einer Menge $C$ mit $\forall i, j \in C : a_{ij} = 0$
- Um alle Werte zu verbessern benötigt man **Färbung** des Adjazenzgraphen $G = (\{1, \ldots, n\}, \{(i,j) \mid a_{ij} \neq 0\})$
- Bei regelmäßigen Matrizen, z.B. Gittern ist eine optimale Färbung oft leicht zu finden
- Abbruchbedingung: **Residuum** $r(k) := b - Ax_k$ ist klein. Zum Beispiel $\|r(k)\|_2 \leq \epsilon \|r(0)\|_2$



### 8.4.2 Successive Overrelaxation (SOR)

Gewichtetes Mittel zwischen altem und neuem Wert

$$x_i(k+1) := (1-\omega)x_i(k) + \frac{\omega}{a_{ii}}(b_i - \sum_{j \neq i} a_{ij} x j)$$

ggf. schnellere Konvergenz. Vor allem für $\omega > 1$!

### 8.4.3 Konjugierte Gradienten

- Für symmetrische positiv definite Matrizen
- Verallgemeingerung: Bikonjugierte Gradienten
- Läßt sich auch als direkter Löser auffassen: (maximal $n$ Iterationen)
- **Präkonditionierung:** Verbessere Konvergenz durch Transformation $\tilde{A} = RAR^T$



### 8.4.4 Multi-Grid Methoden

- Arbeitet mit sukzessiven Vergröberungen des Adjazenzgraphen
- Operationen
  - Relaxation (z.B. Gauss-Seidel)
  - Projektion fein→grob
  - Interpolation grob→fein
- Interpolierte Lösung eines groben Systems kann auch zur Präkonditionierung des feinen Systems verwendet werden.
- In vielen Fällen Ausführungszeit $O(n)$ !
- **algebraische multi-grid**: Eingaben ohne geometrische Interpretation



## Blick über Tellerrand

- Adaptive Gitterverfeinerung
- Unstrukturierte Gitter
- Linearizierung nichtlinearer partieller Differentialgleichungen
- Berechnung von Eigenwerten und Eigenvektoren
- Fast Fourier Transformation



# 9 Sortieren

- $m = nP$ Eingabewerte. Anfangs $n$ pro PE
- u.U. allgemeiner
- Ausgabe global sortiert

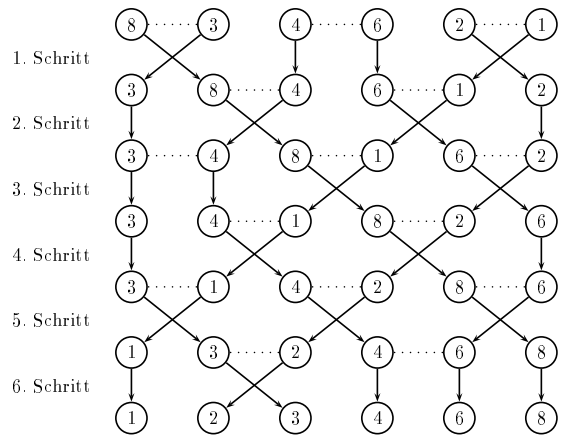
$$\boxed{d_{0,0},\ldots,d_{0,n-1}} \quad , \ldots , \quad \boxed{d_{P-1,0},\ldots,d_{P-1,n-1}}$$

$$\Downarrow \pi$$

$$\boxed{s_{0,0} \leq \cdots \leq s_{0,n_1-1}} \leq \cdots \leq \boxed{s_{P-1,0} \leq \cdots \leq s_{P-1,n_{P-1}-1}}$$

- Vergleichsbasiertes Modell
- $T_{\text{seq}} = T_{\text{compr}} m \log m + O(m)$

## 9.1 Odd-Even-Sort ($n = 1$)



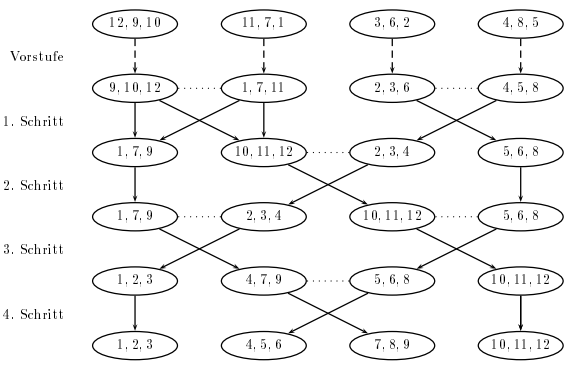



### 9.1.1 Odd-Even-Sort (beliebige $n$)

+ Spezieller Zellularautomat
– $\Theta(mn)$ Vergleiche pro PE
– $\Theta(P)$ bis $\Theta(m)$ Startups
⇒ Nur als Hardwarealgorithmus für kleine $n$ und $P$ (vielleicht) von Interesse



### 9.1.2 Merge-Splitting-Sort




+ Lokal läßt sich effizienter Sortieralgorithmus einsetzen: $\Theta(n \log n)$ Vergleiche
– $P$ Startups
– $\Theta(m)$ Kommunikation pro PE
– Immer noch $\Theta(m)$ Vergleiche für merging
– Effizient nur für $n \gg 2^{PT_{\text{byte}}/T_{\text{compr}}}$



## 9.2 Sortiernetzwerke

- Netzwerk Schaltelemente, die Compare-Exchange machen
+ Einfach
- Bitonic-Sort, ... : Tiefe $O(\log^2 m)$ Größe $O(m \log^2 m)$
- AKS-Sortiernetzwerk: Tiefe $c \log m$, Größe $O(m \log m)$ $c \gg 1000$, ...

## 9.3 Schnelles Ranking ($P = m^2$)

- $m \times m$ logisches Gitter
- Eingabe in PEs $(i, i)$
- $a$ = Broadcast Spalte $i$
- $b$ = Broadcast Zeile $i$
- $x = [a < b]$
- Rang von Element $i$ ist $\sum_j x_{ij}$

\+ $O(\log P)$ Startups

– $\Theta\left(m^2\right)$ Vergleiche insgesamt



## 9.4 Sample-Sort

**Splitter fallen vom Himmel**

{Für $0 < k < P$ sei $p_k$ das Element mit Rang $k \cdot n$.}
{Außerdem setzen wir $p_0 = -\infty$ und $p_P = \infty$.}
initialisiere $P$ leere Nachrichten $N_k$,
$(0 \leq k < P)$;
**for** $i := 0$ **to** $n - 1$ **do**
  bestimme $k$, so daß $p_k \leq d_i < p_{k+1}$;
  nimm $d_i$ in Nachricht $N_k$ auf;
**end for**
**for** $k := 0$ **to** $P - 1$ **do**
  schicke Nachricht $N_k$ an PE $k$;
**end for**
empfange $P$ Nachrichten;
sortiere empfangene Daten;



### 9.4.1 Eine Variante von All-to-all

```
int my_Alltoallv(void *sendbuf, int *sendcounts,
                 int *sdispls, MPI_Datatype sendtype,
                 void *recvbuf, int maxrcount,
                 int *size, int *recvcounts,
                 int *rdispls, MPI_Datatype recvtype,
                 MPI_Comm comm)
{  int numProcs, i;
   MPI_Comm_size(comm, &numProcs);
   MPI_Alltoall(sendcounts, 1, MPI_INT,
                recvcounts, 1, MPI_INT, comm);
   rdispls[0] = 0;
   for(i = 1;  i < numProcs;  i++) {
      rdispls[i] = rdispls[i-1] +
                   recvcounts[i-1];
   }
   *size = rdispls[numProcs-1] +
           recvcounts[numProcs-1];
   return MPI_Alltoallv(sendbuf, sendcounts,
                        sdispls, sendtype,
                        recvbuf, recvcounts,
                        rdispls, recvtype, comm);
}
```

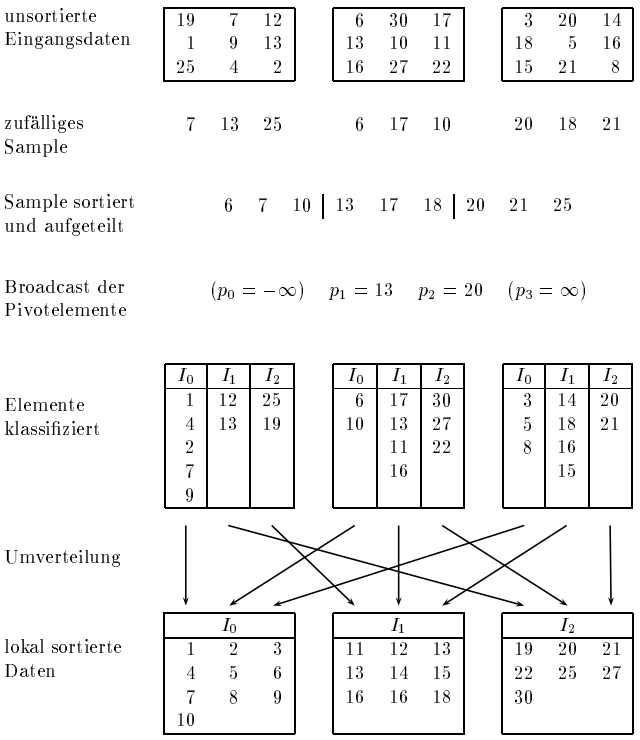


### 9.4.2 Samples

choose a total of $SP$ random elements $s_k$;
($S$ per PE) $(0 \leq k < SP)$;
sort $[s_0, \ldots, s_{SP-1}]$;
**for** $i := 1$ **to** $P - 1$ **do**
  $p_i \leftarrow s_{mi}$;
**end for**
$p_0 \leftarrow -\infty$;  $p_P \leftarrow \infty$;

**Satz 1.** *$S \in O\left(\frac{\log P}{\epsilon^2}\right)$ genügt damit kein PE mehr als $(1+\epsilon)n$ Elemente erhält.*

### Ein Beispiel






### 9.4.3 Samples Sortieren

- Mit `MPI_(All)Gather`
- Schnelles Ranking
- Paralleles Quicksort
- Rekursiv mit Sample-Sort



### 9.4.4 Multi-Merging

- Vorher Sortieren
- Verbesserte Pivot-wahl
- Umverteilung
- $P$-Wege-Mischen



### 9.4.5 Looser Trees

## 9.5 Blick über den Tellerrand

- Datenbanken → parallele Platten
- Geometrische Datenverarbeitung
- Partikelsimulation
- Datenstrukturen: Suchbäume, Prioritätslisten



# 10 Lastverteilung

**Teilproblem:** Eine Berechnung lasse sich parallel abarbeitbare *Teilprobleme* aufteilen

**Lastverteilung:** Was tun wenn Teilprobleme nicht gleich groß sind?

**Klassifikation:** Schwierig weil nur Teilaspekt der Parallelisierung

**Statisch:** Problemunabhängige a priori Zuteilung von Teilproblemen an PEs. Beispiel: Gausselimination zyklische statische Lastverteilung

**Problemabhängig:** Teilproblemgrößen hängen von Eingabe ab. Beispiel: Quicksort

**Dynamisch:** Verteilung zur Laufzeit. Adaptiv.

- Bei nicht vorhersagbaren Teilproblemgrößen
- Bei Clustern mit externer Last

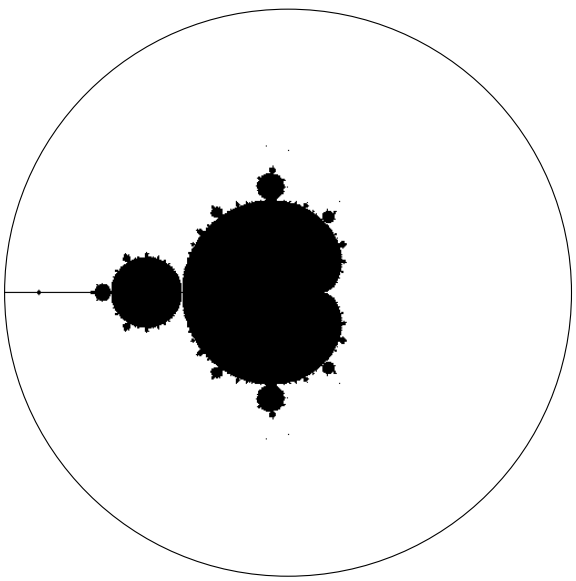


## 10.1 Beispiel Mandelbrotmenge

$$z_c(m) : \mathbb{N} \to \mathbb{C}$$

$$z_c(0) := 0, \quad z_c(m+1) := z_c(m)^2 + c$$

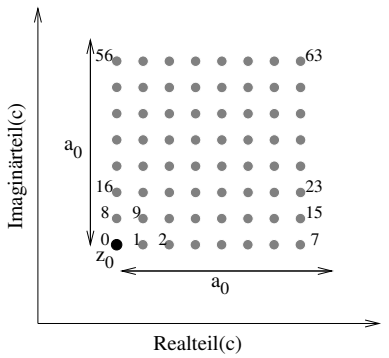
$$M := \{c \in \mathbb{C} \,|\, z_c(m) \text{ ist beschränkt}\} \ .$$





### 10.1.1 Angenäherte Berechnung

- Berechung nur für quadratischen Ausschnitt der komplexen Zahlenebene
- Berechnung nur für diskretes Gitter von Punkten
- $z_c$ unbeschränkt falls $|z_c(k)| \geq 2$
- Abbruch nach $m_{\max}$ Iterationen




Wo liegt das Lastverteilungsproblem?

### 10.1.2 Code

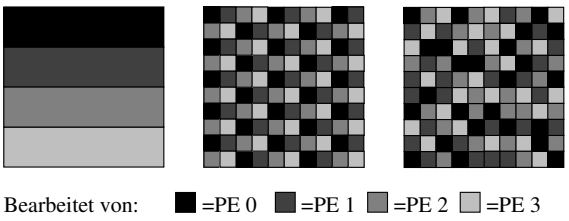
```
int iterate(int pos, int resolution, double step
{
  int iter;
  complex c =
    z0 + complex((double)(pos % resolution)
                 * step,
                 (double)(pos / resolution)
                 * step);
  complex z = c;
  for (iter = 1;
       iter < maxiter && abs(z) <= LARGE;
       iter++)
  {
    z = z*z + c;
  }
  return iter;
}
```



### 10.1.3 Statische Äpfelverteilung

Da kaum Kommunikation stattfindet sind wir sehr flexibel

- Streifenweise Zerlegung
  - Warum attraktiv?
  - Warum besser nicht?
- zyklisch. Gut aber können wir das beweisen?
- Zufällig






## 10.2 Zufällige statische Lastverteilung

- $n$ Teilprobleme
- Größen $l_1, \ldots, l_n$
- Maximale Größe $l_{\max} := \max_{i \le n} l_i$
- Gesamtlast $L := \sum_{i \le n} l_i$
- Zufällige Zuordnung
- $L_i := \sum_j l_j[\text{PE } i \text{ zuständig für } j]$

**Satz 2.**

$$\mathbf{Pr}[\max_i L_i \le (1+\epsilon)\frac{L}{P}] \ge 1 - P^{-\beta}$$

*falls*

$$\frac{L}{P l_{\max}} \ge 2(\beta+1)\frac{\ln P}{\epsilon^2 + \mathrm{O}(\epsilon^3)}$$

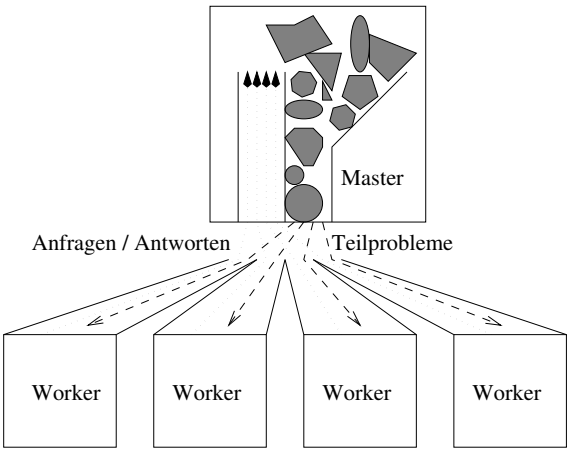


### 10.2.1 Parallelisierung der Zuordnungsphase

- Wenn die Teilprobleme irgendwie auf die PEs verteilt sind: Zufallspermutation via all-to-all. (Siehe auch sample sort)
- Implizites Erzeugen der Einzelteile
  - Teilproblem läßt sich allein aus seiner Nummer $1 \ldots n$ erzeugen.
  - Problem: Parallele Berechnung einer (Pseudo)Zufallspermutation

Zufallsgeneratoren auf der Basis von linearen Kongruenzen lassen sich “vorspulen”.

## 10.3 Das Master-Worker-Schema






### 10.3.1 Diskussion

+ Einfach
+ Natürliches Ein- Ausgabeschema (aber u.U. gesonderter Plattensklave)
+ Leicht zu debuggen
– Kommunikationsengpaß
– Wie soll aufgespalten werden?
– Multilevelschemata sind kompliziert und nur begrenzt hilfreich

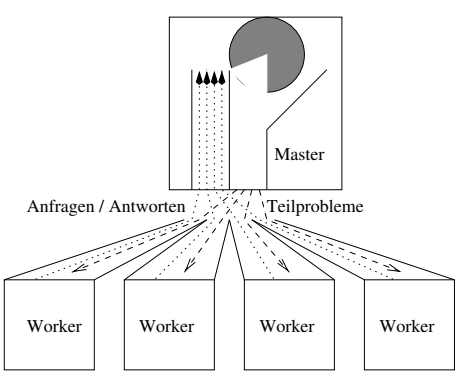


### 10.3.2 Größe der Teilprobleme

- Möglichst grosse Probleme abgeben solange Lastverteilung nicht gefährdet. Warum?
- Konservatives Kriterium: Eine *obere* Schranke für die Größe des abgegebenen Teilproblems sollte $1/P$-tel einer *unteren* Schranke für die sonst im System befindliche Last nicht übersteigen.

– Woher Grössenabschätzung nehmen

- Aggressivere Verfahren ggf. sinnvoll






### 10.3.3 Worker Code

```
do {
  MPI_Receive(.....);
  task = .....;
  workerWork(task);
  MPI_Send(....); /* result and request */
} while (.....); /* not finished */
```

### 10.3.4 Einfacher Master Code ($P = 2$)

```
newWorkForWorker(0);
newWorkForWorker(1);
while (.....) {
  MPI_Recv(....., MPI_ANY_SOURCE, .....);
  .....
  /* process results and send new work */
}
```



### 10.3.5 Fairer (aber langsamer) Master Code

```
newWorkForWorker(0);
newWorkForWorker(1);
MPI_Irecv(....., &requests[0]);
MPI_Irecv(....., &requests[1]);
while (.....) {
   MPI_Waitsome(2, requests, &outcount,
                indices, statuses);
   for (i=0;  i < outcount;  i++) {
      /* process results and
         send new work for
         worker indices[i] */
      .....
      /* post new receive operation */
      MPI_Irecv(.....,
                &requests[indices[i]]);
   }
}
```



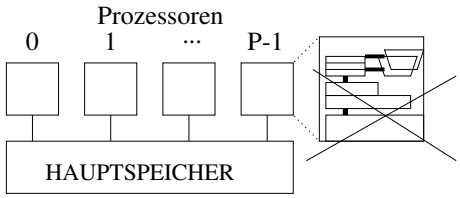
## 10.4 Blick über den Tellerrand

- Baumsuche
  - Prioritsierte Last (Branch-and-Bound)
  - Spielbaumsuche (Schach ... )
- Job Scheduling (Betriebssystemsichtweise)
- Partielle DGLS
  - Unstrukturierte ggf. adaptiv verfeinerte Gitter
  - Komplizierte Randbedingenen (Verbrennungsmotor, ... )
  - Schrittweitenanpassung (steife DGL)
  - Komplizierte Modelle: Strömung, Verbrennung, Partikel, ... (Beipiele Klimamodellierung→, Verbrennungsmotor)
- Partikelsimulation: Particle in cell, ...



# 11 Parallelverarbeitung mit gemeinsamem Speicher

## PRAM




- Jeder kann in konstanter Zeit beliebige Speicherstelle zugreifen
- Synchron

**EREW** Exclusive Read Exclusive Write

**CREW** Concurrent Read Exclusive Write

**CRCW** Concurrent Read Concurrent Write

**arbitrary** Einer gewinnt

**common** Alle müssen sich einig sein

...

## Diskussion

+ Parallele Algorithmen leicht formulierbar
– Physikalisch realisierbar?
- SBPRAM, Tera MTA: Jein
  – Verbindungsnetzwerk der Tiefe $O(\log P)$
  – Jeder physikalische Prozessor nimmt $O(\log P)$ simulierte PEs round robin dran
  + Zugriffslatenz wird verborgen
  – Aufwendige Hardware
  ? Kann man die schnell genug machen?
  – Man braucht viel Parallelismus
  + Aber der darf sehr feinkörnig sein



## 11.1 Symmetric MultiProcessors

Unterschiede zur PRAM

- Lokale Caches
- Asynchron
  + Flexibler
  – Schwerer zu programmieren

Beispiele:

- 2× PentiumII (unser Cluster) über (schwächlichen) Bus gekoppelt
- Sun E10000 mit bis zu 64 UltraSparcII (ehemals Cray) Crossbar-Architektur ... demnächst in diesem Kino (16 PEs)
- NEC SX-5 Node bis zu 16 Vektorprozessoren a 4GFlop



### 11.1.1 Fehlt da was?

Tradeoff Implmentierbarkeit versus Funktionalität

- Thread-Unterstützung ist optional
- Nichtblockierende kollektive Operationen
- “Active Messages”
- Kanäle und andere persistente Operationen
- Fairness



# 12 POSIX Threads

**P**ortable **O**perating **S**ystem **I**nterface standard for Uni**x**

Was is ein Thread (Aktivitätsfaden)?

**Prozess** besteht aus

- Adreßraum mit Code globalen Variablen, ...
- Dateien
- Einem oder mehreren Threads

**Thread** besteht aus

- Stapel
- Registerbelegung

## 12.1 Überblick

- Thread Erzeugung
- Synchronisationsprimitive:
  - Gegenseitiger Ausschluß
  - Bedingungsvariablen

**Hier nicht**

- Keys
- Signale
- Scheduling
- Beenden, Warten, Cancel, ... (Warum?)



## 12.2 Was braucht man

- `#include <pthread.h>`
- `g++ ... -lpthread -lposix4`



## 12.3 Threads Erzeugen

Ein Prozeß hat per default einen Thread, der mit `main()` beginnt.

```
int pthread_create(
            pthread_t      *new_thread_ID,
      const pthread_attr_t *attr,
            void * (*start_func)(void *),
            void *arg);
```

`start_func` Womit fängt er an

`arg` Parameter für ihn

`new_thread_ID` Zum "Wiederekennen"

`attr` Wie verhält er sich, was kann er

Parallele Threads:

```
pthread_attr_init(&attr);
pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTE
pthread_create(&tid, &attr, start_func, arg);
```



### 12.3.1 Semantik

- Globale Variablen gibt nur einmal → impliziet Parameterübergabe über globale Variablen ist gefährlich
- `static` Variablen ebenfalls!
- lokale (`auto`) Variablen sind lokal (stack, register)
- Jeder kann dynamisch Speicher allozieren
- Standard Library ist threadsafe aber es gibt jedes File nur einmal (`stdout` etc!)

## 12.4 Gegenseitiger Ausschluß

```
int pthread_mutex_init(pthread_mutex_t *mp,
    const pthread_mutexattr_t *attr);
int pthread_mutex_lock(pthread_mutex_t *mp);
int pthread_mutex_trylock(pthread_mutex_t *mp);
int pthread_mutex_unlock(pthread_mutex_t *mp);
```

`lock` Durch die Tür gehen und hinter sich abschließen oder warten bis jemand rauskommt (britisch in line)

`try_lock` Schauen ob Tür offen ist

`unlock` Aufschließen, rausgehen Tür offen lassen



## Beispiel Zähler

```
int sharedCounter = 0;
pthread_mutex_t sharedCounterM;
...initialize somewhere...

int fetchAndIncrement()
{
  pthread_mutex_lock(&sharedCounterM);
  int result = sharedCounter;
  result++;
  sharedCounter = result;
  pthread_mutex_unlock(&sharedCounterM);
  return result;
}
```

Dünnbesetzte Matrizen, Mandelbrotmenge, Master-Worker, ...



## 12.5 Bedingungsvariablen:

```
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
int pthread_cond_timedwait(pthread_cond_t *cond,
                           pthread_mutex_t *mutex,
                 const struct timespec *abstime);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond)
```

Analogie Schranke an der Unieinfahrt:

`wait` Warte auf Freigabe

`timedwait` mit begrenzter Geduld

`signal` Nächsten durchlassen

`broadcast` Alle durchlassen

Beispiel: Barrier implementieren



## 12.6 Einsatzzwecke

- Interaktive Programme (nicht hier)
- Asynchrones I/O (nicht hier)
- Primäres Werkzeug zur Parallelisierung (eine shared memory Maschine)
- Lokale Interaktion bei Clustern von shared memory Maschinen
- Entkopplung von Kommunikation und Berechnung Beispiel: Parallele Priority Queue und Branch-and-Bound

## 12.7 Vergleich mit MPI

+ Speicherzugriff hat geringere Latenz als Kommunikation (Warum?)
+ Eingaben verteilen ist fast geschenkt (Auch schnell?)
+ Speicherzugriff braucht keine Umständlichen Prozeduraufrufe $\rightarrow$ oft einfacher und flexibler
+ Locks etc. sollten (!?) schneller als $T_{\text{start}}$ sein.
– Low Level
  – Aliasing
  – Einfache Lösungen sind oft nicht skalierbar (Beispiel Zähler)
  – Keine kollektiven Operationen
  – Hohe **Bandbreiten** können Tricks erfordern, Cache line alignment, ...
  – Vielfä(lt)(d)ige Fehlermöglichkeiten
– Erfordert spezielle Hardware
– Skalierbarkeit umstritten



## 12.8 Andere Programmierumgebungen für gemeinsamen Speicher

**Programmiersprachen:** Concurrent Pascal, Modula-II, Ada, OCCAM (?!)

**Libraries:** objektorientiert, ...

**OPENMP:** Compiler `pragmas`. Idee: erhalte Semantik des seq. Programms

**Compiler:** (halb)automatische Parallelisierung

**MPI-2** ?



# 13 MPI-2

- Einseitige Kommunikation "SM Surrogat"
- Prozeßverwaltung
- Parallele Ein- und Ausgabe
- Kooperation mit Threads
- Erweiterte kollektive Operationen
- C++ Binding
- MPI-3 ?



## 13.1 Einseitige Kommunikation

- Gegensatz zu "Zweiseitig"
- *Ursprungs*-PE (Sender oder Empfänger) entscheidet *wann* und *wohin* übertragen wird

+ Schneller (vor allem bei SM) und
+ Flexibler als zweiseitig

### 13.1.1 Ein Fenster zur Welt

`MPI_Alloc`: liefert Speicher der besonders geeignet ist. U.u. auch für zweiseitige Kommunikation

`MPI_Win_Create`: Jedes PE in einem Kommunikator macht ein Fenster in seinen Speicher global bekannt. Das

**Fenster** übernimmt danach die Rolle dieses Kommunikators (für einseitige Kommunikation)



### 13.1.2 Semantisches Modell

- Private Fensterkopie
- Öffentliche Fensterkopie
- lokale Zugriffe verändern private Kopie
- Auswirkung auf öffentliche Kopie implementierungsabhängig



### 13.1.3 Get

```
int MPI_Get(
  void *origAdr, int origCnt,
  MPI_Datatype origType,
  int targRank,
  MPI_Aint targDisp, int targCnt,
  MPI_Datatype targType, MPI_Win win);
```

verhält sich ähnlich wie

```
MPI_Irecv(origAdr, origCnt, origType,
          targRank, MPI_ANY_TAG, comm, .....);
```

zusammen mit einem

```
MPI_Isend(
  base + targDisp*disp_unit, targCnt, targType,
  origRank, 42, comm);
```

auf der Gegenseite.

- Zugriff auf öffentliche Kopie von PE `targRank`



### 13.1.4 Put

- Analog zu Get
- Zugriff auf öffentliche Kopie aber
- Lokale Kopie *kann* beeinflußt werden

### 13.1.5 Accumulate

- Put mit
- (vordefinierter) Reduktionsoperation (`MPI_op`) z.B. Addition
- MPI garantiert Atomizität bei parallelen Zugriffen

### 13.1.6 `MPI_Win_fence(assert, win)`

- Kollektiv auf `win`
- Bring alle Operationen auf `win` zum Abschluss
- Lokale Zugriffe werden öffentlich sichtbar
- Put/Get Zugriffe werden lokal sichtbar
- Barrier (wenn `assert`$\neq 0$)



### 13.1.7 Locks

```
int MPI_Win_lock(int type, int rank, int assert,
                 MPI_Win win);
int MPI_Win_unlock(int rank, MPI_Win win);
```

Lock

- Wie Posix-Thread Mutex auf *ganzes* Fenster `win` von PE `rank`
- Lokale Zugriffe werden öffentlich sichtbar
- `type`: `MPI_LOCK_EXCLUSIVE`, `MPI_LOCK_SHARED`

Unlock

- Freigabe
- Einseitige Kommunikationen auf Ursprung und Ziel beenden
- Abgleich private, öffentliche Kopie



### 13.1.8 Vergleich mit POSIX Threads

+ Mehr Funktionalität
+ Auch auf distributed Memory
– Umständlich
– In Praxi vermutlich langsamer (Compilerunterstützung?)



## 13.2 Prozeßverwaltung

### 13.2.1 `MPI_Comm_Spawn`

- Erzeugt neue Prozesse mit eigenem `MPI_COMM_WORLD`
- Interaktion mit anderen "Welten" über *Interkommunikator*

### 13.2.2 ...

## 13.3 Parallele Ein- und Ausgabe

- Files sind Arrays
- Die Sicht (view) eines Prozesses auf eine Datei wird durch einen MPI-Datentyp festgelegt
- Sichten können sich überlappen oder ergänzen
- Blockierender und Nichblockierender Zugriff
- Kollektiver und individueller Zugriff (einschließlich einfacher Form von nichblockierendem kollektiven Zugriff!)
- Shared file pointer erlaubt implizite Master-Worker-Lastverteilung



### 13.3.1 Fehlt da was?

Tradeoff Implementierbarkeit versus Funktionalität

- Thread-Unterstützung ist optional
- Nichtblockierende kollektive Operationen
- "Active Messages"
- Kanäle und andere persistente Operationen
- Fairness



# 14 Fehlersuche in Parallelen Programmen

## 14.1 Eigentlich nichts neues

- Schwer weil Fehlersuche immer schwer.
- Aber nicht viel schwerer: (im Prinzip)
  - Der parallele Anteil ist oft klein
  - SPMD macht Kommunikationsmuster übersichtlich
  - Nachrichtenkopplung macht Interaktion und Synchronisation explizit



## 14.2 „Langweilige" Schwierigkeiten

**Häufige Programmierfehler**

- Werden `MPI_Init` und `MPI_Finalize` richtig verwendet?
- Puffer richtig dimensioniert und übergeben?
- Einigkeit über Typ und Länge einer Nachricht?
- Tags verwechselt
- kollektive Operation *überall* und *konsistent* aufgerufen? (root, ... )

**Beim Testen**

- Ein/Ausgabe unübersichtlich
- Bei Absturz gehen ggf. Ausgaben verloren!
- Debuggerläufe werden noch schneller unüberschaubar (selbst wenn möglich)

## 14.3 Defensives Programmieren

- `if` und `switch` vermeiden, ... (siehe auch unten)
- Reichlich Assertions. Vorteil: Keine Belästigung ohne Fehler $\rightsquigarrow$ „Programming by Contract“; Vorbedingungen, Nachbedingungen, Schleifen(in)varianten, Datenstrukturinvarianten
- Kommunikation und Berechnung logisch trennen
- Handler-Routinen zur Behandlung verschiedener Nachrichttentypen ($\rightsquigarrow$ OO-parallele Sprachen, aktive Objekte, Aktor-Sprachen, active Messages) Beispiel: Zufälliges Anfragen
- Lieber zuviele Tags



**Der Fall $P = 1$**

- Kollektive Operationen OK
- `MPI_ISend`, `MPI_BSend` vermutlich OK.
- `MPI_Send` funktioniert nicht immer.
- `MPI_SSend` kann nicht funktionieren.
- Master-Worker: Master sollte auch Worker sein können.



## 14.4 Einfache Maßnahmen beim Testen

- Debug-Optionen der MPI-Implementierung aktivieren (ausführliche Fehlermeldungen etc.)
- Alle Assertions aktivieren (routinemäßig)
- Ausgabeströme auseinanderhalten:
  - Mindestens zeilenweise Pufferung (u.U. nicht default bei `stderr`)
  - Ggf. autom. PE-Nummer-Prefix
  - Bei Absturz gehen ggf. Ausgaben verloren!
  - Im Zweifel `MPI_Gather` auf Ausgaben
- Eins, zwei, unendlich
- Landen wirklich alle an den vorgesehenen Synchronisationspunkten (Barrier?)



## 14.5 „Wissenschaftlich satisfaktionsfähige“ Schwierigkeiten

**Deadlock:** (zyklisches Warten auf Ressourcen) $\rightsquigarrow$ irgendwas hängt

**Lifelock:** Nachrichten kommen nicht durch, Terminierung wird nicht festgestellt, ... (darum wünscht man sich Fairness)

**Race Condition:** Fehler treten nur bei bestimmter zeitlicher Abfolge von Ereignissen auf. $\Rightarrow$ schwer reproduzierbar

## 14.6 Performance Debugging

**Modellieren oder Messen**

- Anzahl Nachrichten OK?
- Gesamtdatenmenge OK?
- Wartezeiten?
- Flaschenhälse?
- Länge der kritischen Pfade?

Es gibt grafische Werkzeuge mit den gleichen Vor- und Nachteilen wie parallele Debugger. Zusätzlich verfälscht die Messung bei großem $P$ und kleinen Nachrichten die Leistung



# 15 Ausgewählte Algorithmen für kollektive Kommunikation

## 15.1 All-to-All für lange Nachrichten

Nachrichten $M_0, \ldots, M_{P-1}$ pro PE

**for** $j := 1$ **to** $P/2 - 1$ **do**
  **for** $i := 0$ **to** $P - 1$ **dopar**
    Exchange messages $M_{i+j \bmod P}$ and $M_i$
    between PEs $i + j \bmod P$ and $i$

Komplikationen

- Gitter, ...
- Unregelmäßige Nachrichtenlängen: Üblich Technik ist, Kleinhächseln, an zufällige Adresse, dann erst an Endadresse



## 15.2 All-Gather für kleine Nachrichtenlängen

[KumarEtAl Program 3.6] Jedes PE hat Nachricht $M$. PE-Nummer $j$

$M_{\text{all}} := M$
**for** $i := 0$ **to** $\log P - 1$ **do**
  $M := M_{\text{all}}$ from PE $i \oplus 2^j$  (* SendRecv *)
  $M_{\text{all}} := \text{concat}(M_{\text{all}}, M)$

- Interpretation: $P$ verschachtelte Broadcasts
- Kompromisse bei mittleren Nachrichtenlängen
- Analog All-to-all, All-reduce, (All-scan)