Paralleles Höchstleistungsrechnen

Peter Bastian
Interdisziplinäres Zentrum für Wissenschaftliches Rechnen
Universität Heidelberg
Im Neuenheimer Feld 368
D-69120 Heidelberg
email: Peter.Bastian@iwr.uni-heidelberg.de

5. November 2008

Inhaltsverzeichnis

1 Motivation

1.1 Organisatorisches

Vorlesung

- Dozent
  - Peter Bastian
  - zu finden in: INF 368, Raum 420
  - email: `Peter.Bastian@iwr.uni-heidelberg.de`
- Vorlesung
  - Di 9-11 in 368/532, Do 9-11 in 350/U014
  - Skript, Folien (falls verfügbar), Übungsaufgaben: `http://hal.iwr.uni-heidelberg.de/phlr.html`

Übung

- Leiter: Christian Engwer
- findet statt im CIP-Pool des IWR, Otto-Meierhof-Zentrum
- Zeit: **Noch zu vereinbaren!**
- **Anmeldung zur Übung**
- Schein auf Übungsaufgaben
- Bachelor/Master: mündliche Prüfung (bzw. Klausur bei großem Andrang)
- Voraussetzung für die Teilnahme an der mündlichen Prüfung/Klausur: 50% der Übungsaufgaben
- C/C++ Kentnisse erforderlich

1.2 Motivation

Warum parallel Rechnen ?

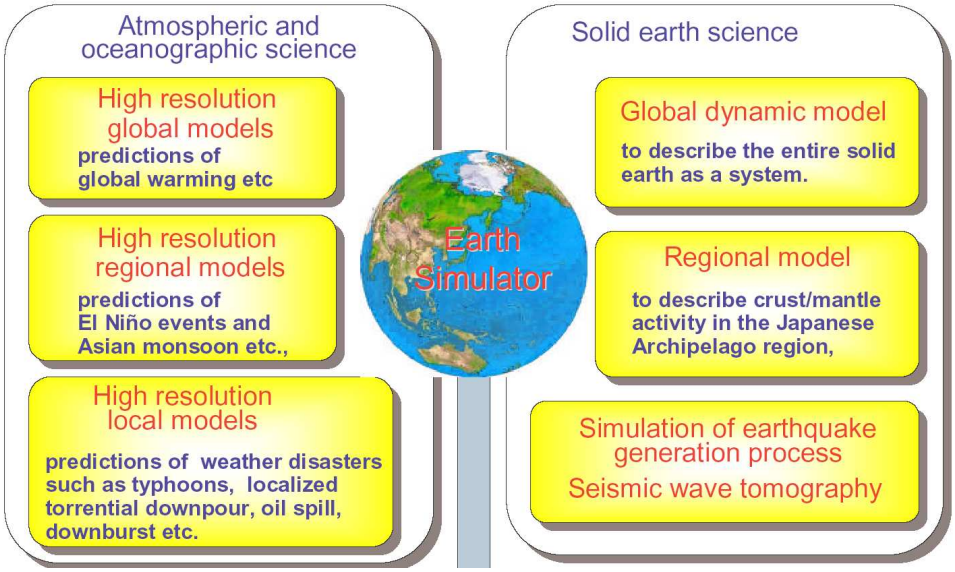
- Nebenläufigkeit
  - Abarbeitung mehrerer Prozesse auf einem Prozessor
  - Multi-Tasking Betriebssysteme seit den 60er Jahren
  - Bedienung mehrerer Geräte und Benutzer
  - Ziel: Steigerung der Auslastung
  - "Hyperthreading": Nutze Wartezeiten des Prozessors
  - Mehrere Dinge gleichzeitig : Web-Browser, Desktop
  - Koordinationsproblematik tritt bereits hier auf
- Verteilte Anwendungen

  - Datenbasis ist inhärent verteilt: betriebswirtschaftliche Software, Warenfluß in großen Unternehmen
  - Hier wichtig: plattformübergreifende Kommunikation, Client-Server Architekturen
  - Auch wichtig: Sicherheit, VPN, etc. (behandeln wir nicht)

High Performance Computing

- Treibt Rechnerentwicklung seit Anfang 1940er
- Anwendungen: mathematische Modellierung und numerische Simulation
- ungestillter "Hunger" nach Rechenzeit:
  - Modellfehler → detailliertere Physik
  - Approximationsfehler → mehr Freiheitsgrade
- Grand Challenges:
  - Kosmologie, z. B. Galaxiendynamik
  - Proteinfaltung
  - Erdbebenvorhersage
  - Klimaentwicklung
- ASCI Program (Advanced Simulation and Computing) Programm amerikanischer Parallelrechner seit 1992, Funding 2004: 200 Mio US-$ DoE, 87 Mio US-$ NSF

Earth Simulator (Japan)



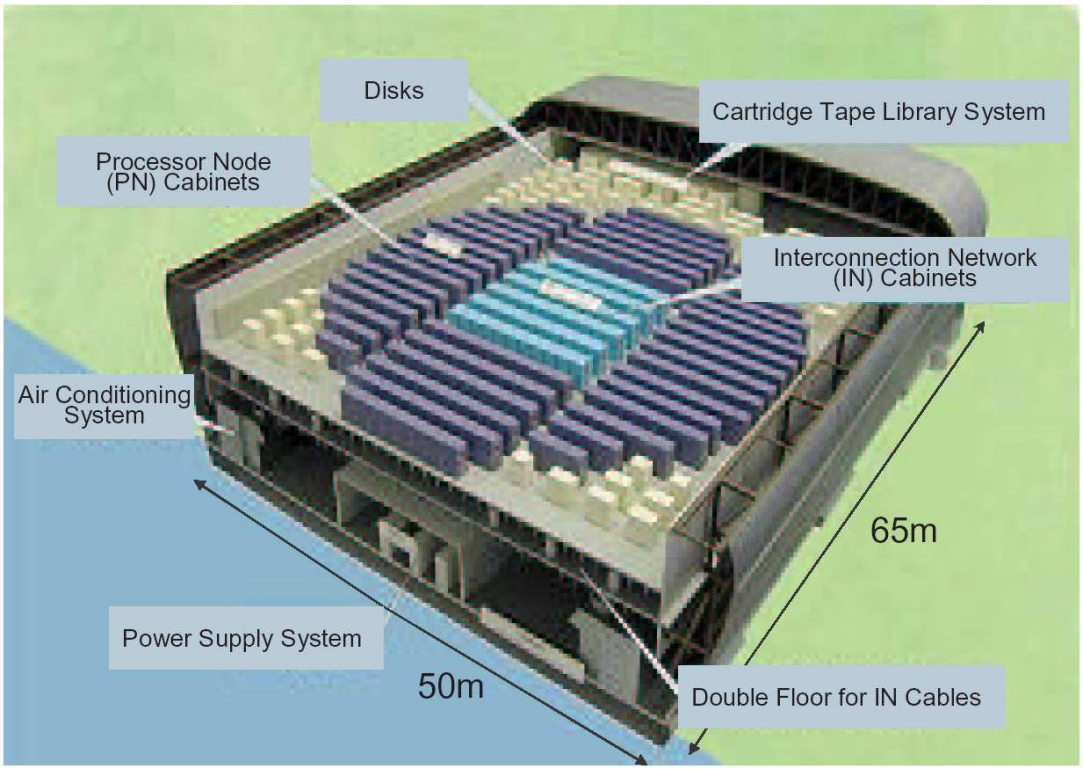
- Juni 2002 bis Juni 2004 der leistungsfähigste Computer der Welt
- Kosten: 350 bis 500 Mio US-$
- Folie (und die folgenden) von J. Dongarra

Earth Simulator Architecture

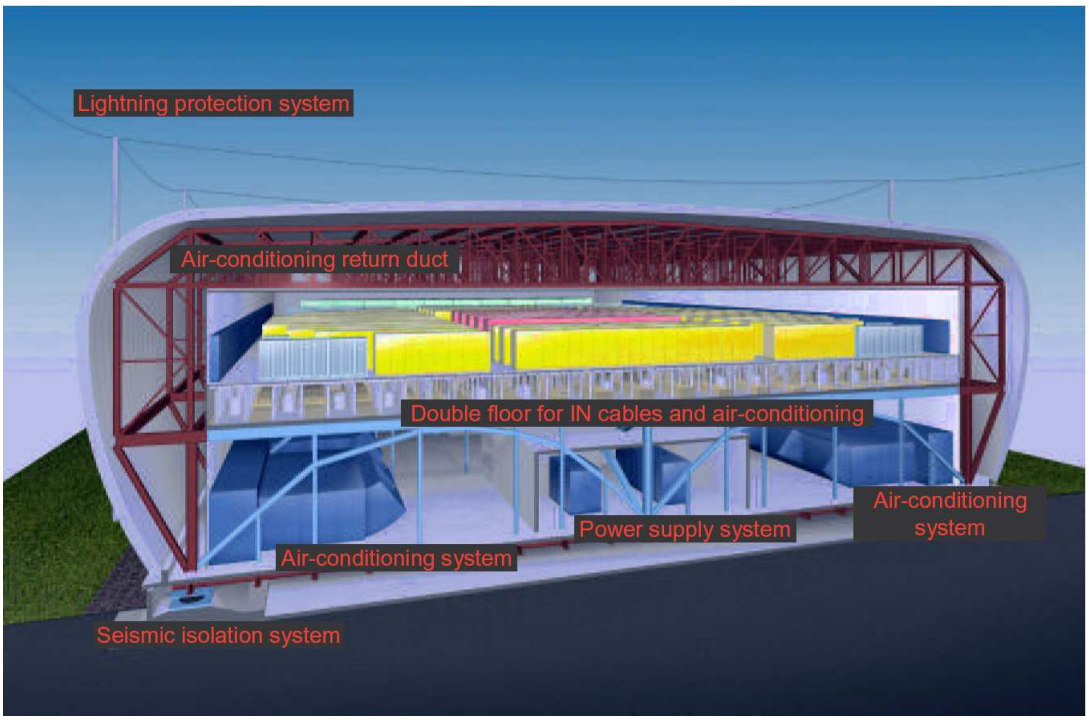
Earth Simulator

- Based on the NEC SX architecture, 640 nodes, each node with 8 vector processors (8 Gflop/s peak per processor), 2 ns cycle time, 16GB shared memory.
  - Total of 5104 total processors, 40 TFlop/s peak, and 10 TB memory.
- It has a single stage crossbar (1800 miles of cable) 83,000 copper cables, 16 GB/s cross section bandwidth.
- 700 TB disk space
- 1.6 PB mass store
- Area of computer = 4 tennis courts, 3 floors

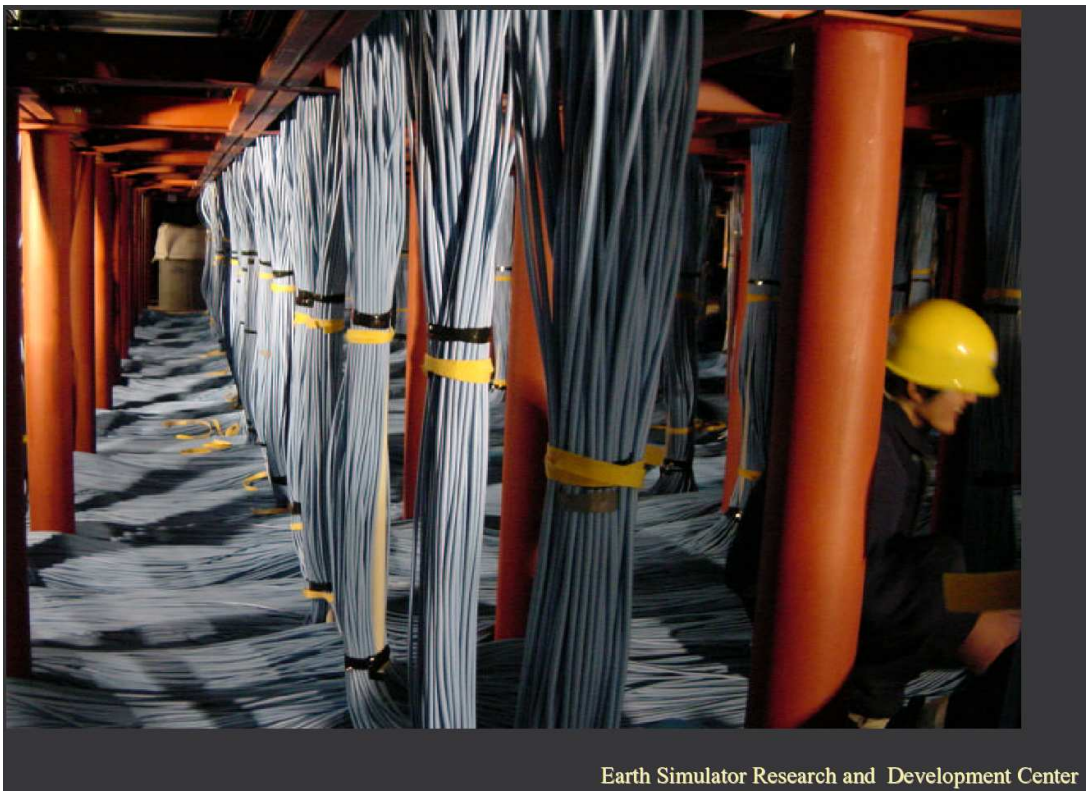
Earth Simulator Gebäude



Earth Simulator Gebäude

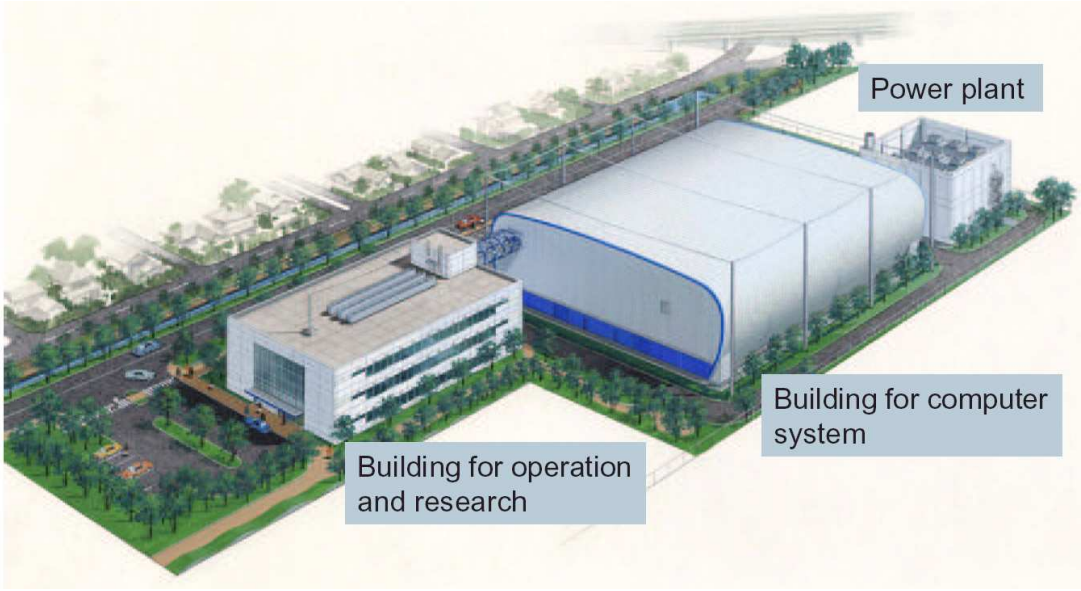


... die Kabel



... die Kabel

... und ein Kraftwerk

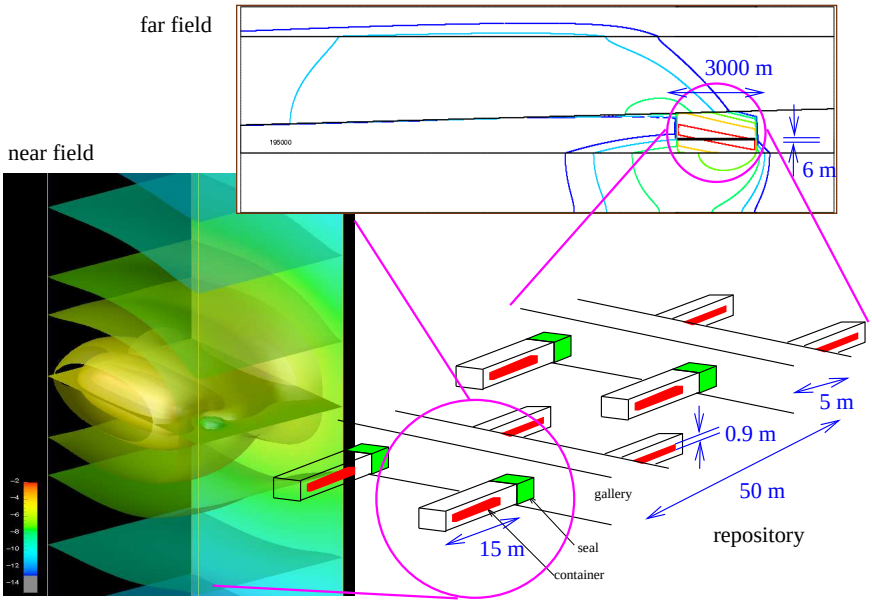


Terascale Simulation Facility

- Lawrence Livermore National Laboratory
- BlueGene/L 212992 Prozessoren, 478 TFLOPS seit 2007

1.3 Einige Beispielanwendungen

Couplex Benchmark



Couplex1, Couplex2

Couplex Transport Model

Flow equation:

$$\nabla \cdot u = f \quad \text{in } \Omega, \qquad\qquad u = -K\nabla H$$

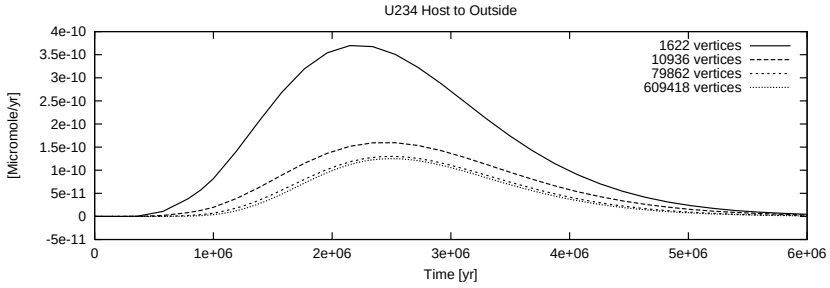
Transport equations i = silica, ^{135}Cs, ^{238}Pu, ^{242}Pu, ^{234}U, ^{238}U:

$$R\omega\left(\frac{\partial C_i}{\partial t} + \lambda C_i\right) + \nabla \cdot j_i = q(C_1, \ldots, C_n) \qquad \text{in } \Omega,$$
$$j_i = uC_i - D(u)\nabla C_i$$

Processes:

- Convection-dispersion (Scheidegger)
- Radioactive decay ^{238}Pu$\rightarrow^{234}$U, ^{242}Pu$\rightarrow^{238}$U
- Precipitation: ^{238}Pu, ^{242}Pu, ^{234}U, ^{238}U as disolved components and (immobile) solid phase

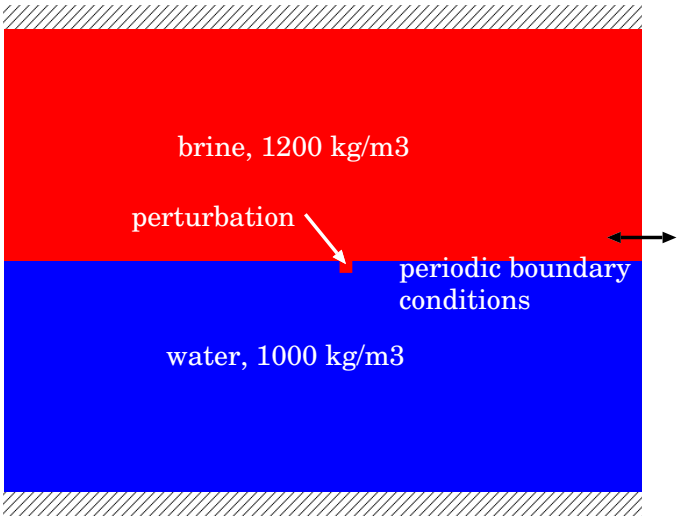
Couplex 2



P	Unbekannte	Par. Rechenzeit [h]	Seq. Rechenzeit [h]
1	120296	2.4	2.4
8	878482	6.3	21.8
64	6703598	6.4	226.3

(Gemeinsame Arbeit mit Stefan Lang)

Dichteströmung



Salzstock, Ozean, Atmosphäre, Sterninneres, Erdmantel, Kaffeetasse

Density driven flow in Porous Media

- Flow equation (Bussinesq approximation):

$$\nabla \cdot u = f, \quad u = -\frac{K}{\mu}(\nabla p - \varrho(C)g), \quad \varrho(C) = C\varrho_b + (1-C)\varrho_0 \quad \text{in } \Omega$$

- Transport equation:

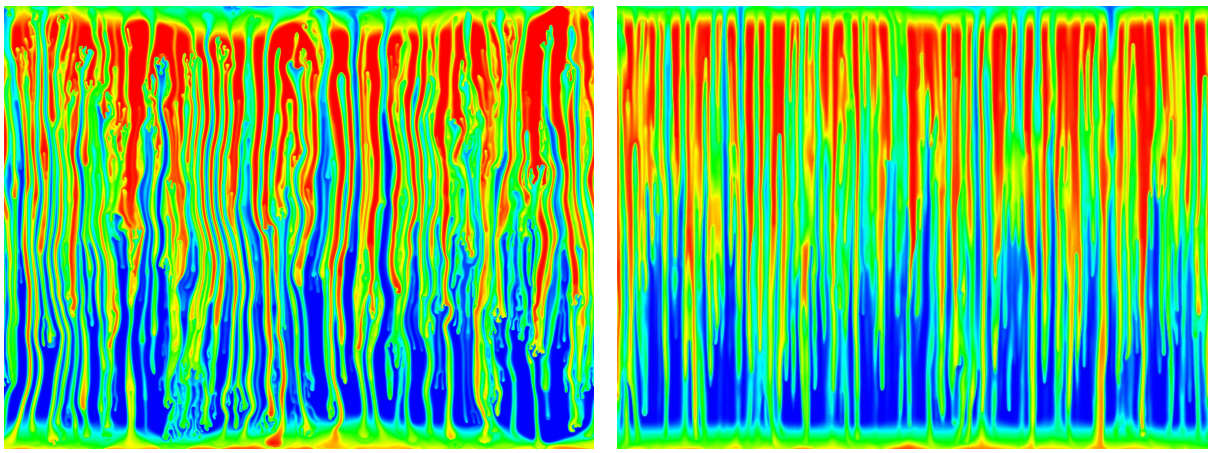
$$\frac{\partial(\Phi C)}{\partial t} + \nabla \cdot j + q = 0, \quad j = Cu - D\nabla C$$

- *Elliptic* flow equation coupled to *parabolic/hyperbolic* transport equation
- Cell centered finite volume discretization, second order Godunov scheme for convective terms
- Cell centered multigrid method
- Decoupled solution scheme

Simulation Results

Comparison of two-dimensional and three-dimensional simulation

2d 3d

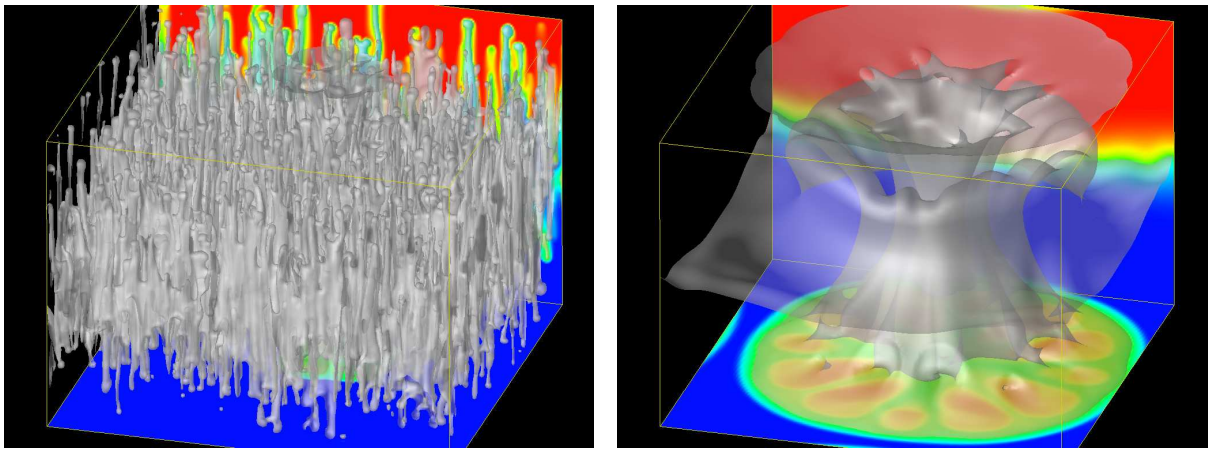


Simulation Data

- Grid size: $1024 \times 1024 \times 768$, about 9000 time steps
- Solves linear system with $8 \cdot 10^8$ unknowns per time step (requires 30 seconds on 384 processors)

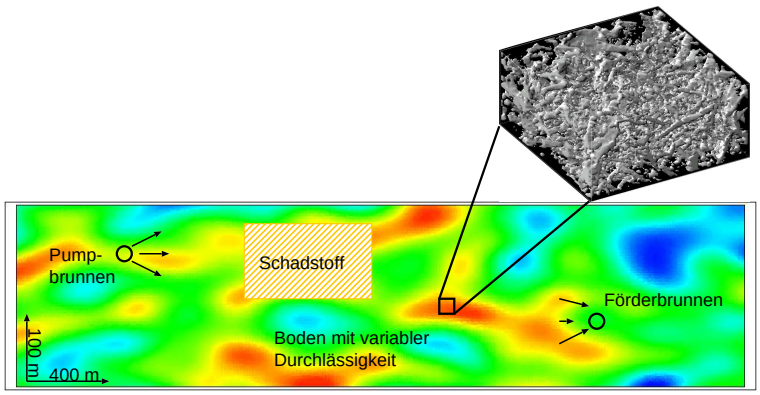
- Data produced per time step: ≈ 3.2 GByte in binary float
- Data produced by whole simulation: ≈ 29 TByte
- Visualization requires intelligent ways of data reduction and/or parallelized visualization tools
- In addition I/O capabilities of HELICS should be improved in the future
- Fault-tolerance might become an issue

Virtuelles Labor

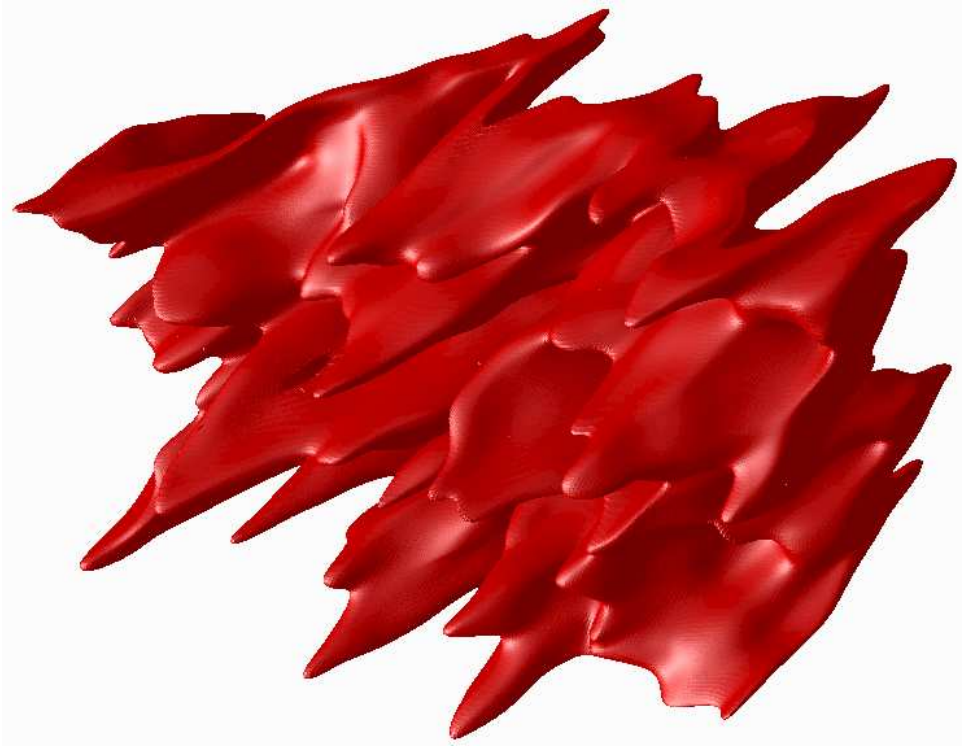


- Verändere physikalische Parameter (links wenig Diffusion, rechts viel Diffusion)
- Visualisierung großer Datenmengen ist ein Problem
- Alle Komponenten einer Simulation müssen parallel bearbeitet werden

In-Situ Reinigungsverfahren



3D Transport in Heterogenem Medium

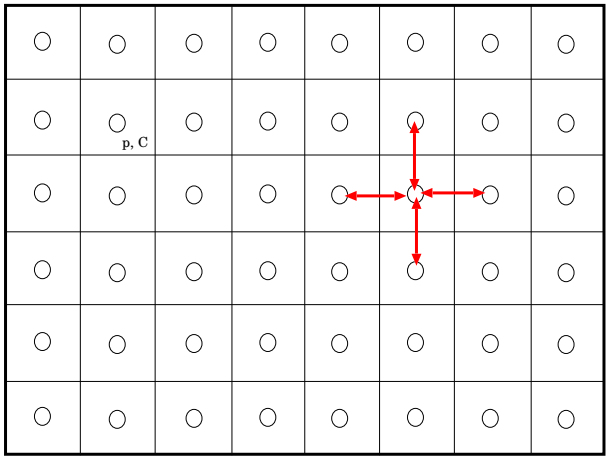


Zellenzentrierte Finite-Volumen Verfahren (CCMG, HOG)

$4604 \times 384 \times 256 \approx 4.5 \cdot 10^8$ Zellen, $\approx 3 \cdot 10^4$ time steps (Transparenz)

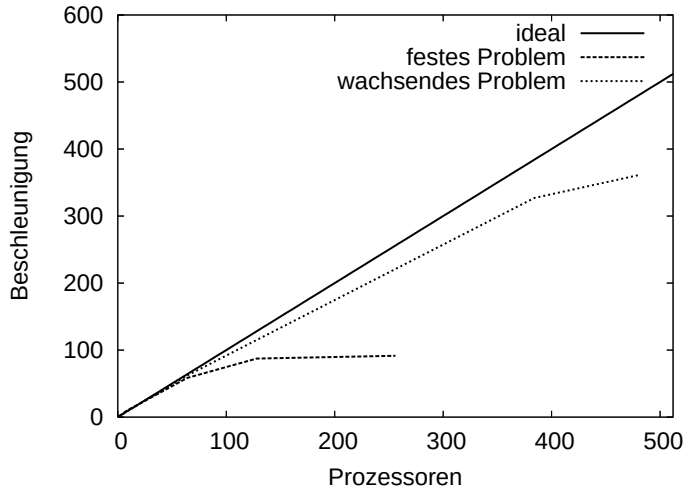
Parallele Rechnung auf 384 Prozessoren

Numerische Strömungsberechnung



- Pro Zelle: Phys. Größen wie Druck, Temperatur, Konzentration
- Je mehr Zellen desto genauer das Ergebnis
- Lokale Datenabhängigkeiten (Differentiation!)

Beschleunigung



- Gegebenes Problem immer schneller rechnen
- Mit mehr Prozessoren immer größere Probleme rechnen

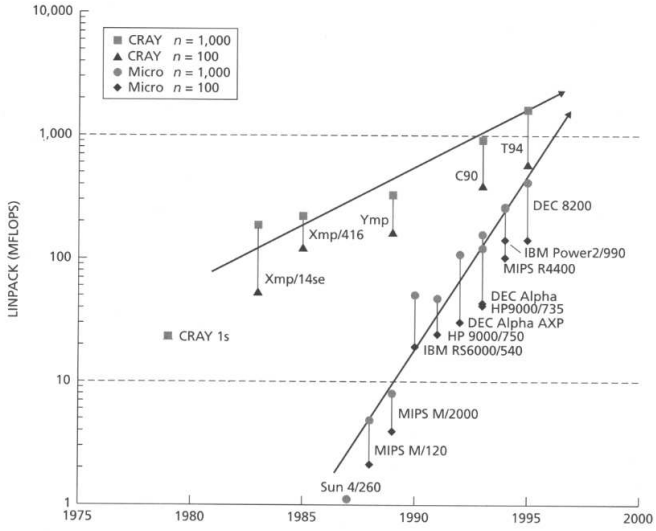
1.4 Superrechner

Was ist ein Superrechner ?

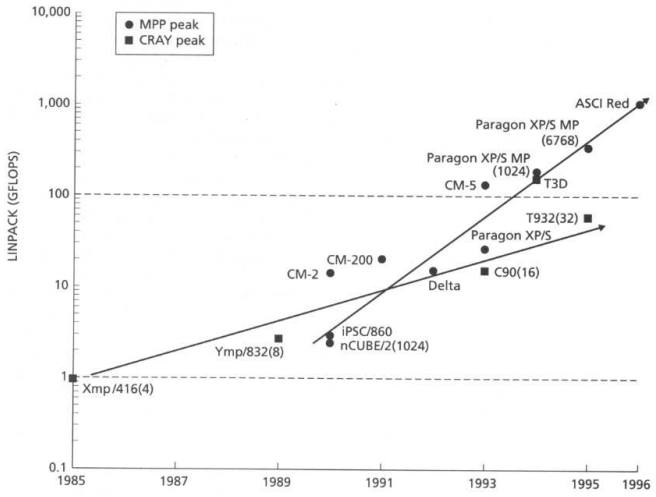
Gerät	Jahr	$	Takt [MHz]	MBytes	MFLOPS
CDC 6600	1964	7M $	10	0.5	3.3
Cray 1A	1976	8M $	80	8	20
Cray X-MP/48	1986	15M $	118	64	220
C90	1996		250	2048	5000
ASCI Red	1997		220	1212000	2388000
Pentium 4	2002	1500	2400	1000	1000
Core 2 XE QX6800	2007	2000	2933	>4000	25400

- X-MP/48: P=4, C90: P=16, ASCI Red: P=9152
- PC von heute = Superrechner von gestern

Entwicklung Einzelprozessoren in den 90ern[1]



Entwicklung Multiprozessoren[2]



HELICS

Entwicklung seit den 90er Jahren: Cluster Computer

[1] Culler/Singh, Parallel Computer Architecture.
[2] Culler/Singh, Parallel Computer Architecture.

Aufbau aus Standardbauteilen

Aktuelle Entwicklung

- Multicore Prozessoren
- Coprozessoren Cell, Clearspeed
- General Purpose GPU-Computing
- Petaflop Computer: 10^{18} Gleitkommaoperationen pro Sekunde erreicht im Juni 2008 der IBM Roadrunner
- 122400 Cores, Opteron mit Cell Coprozessoren

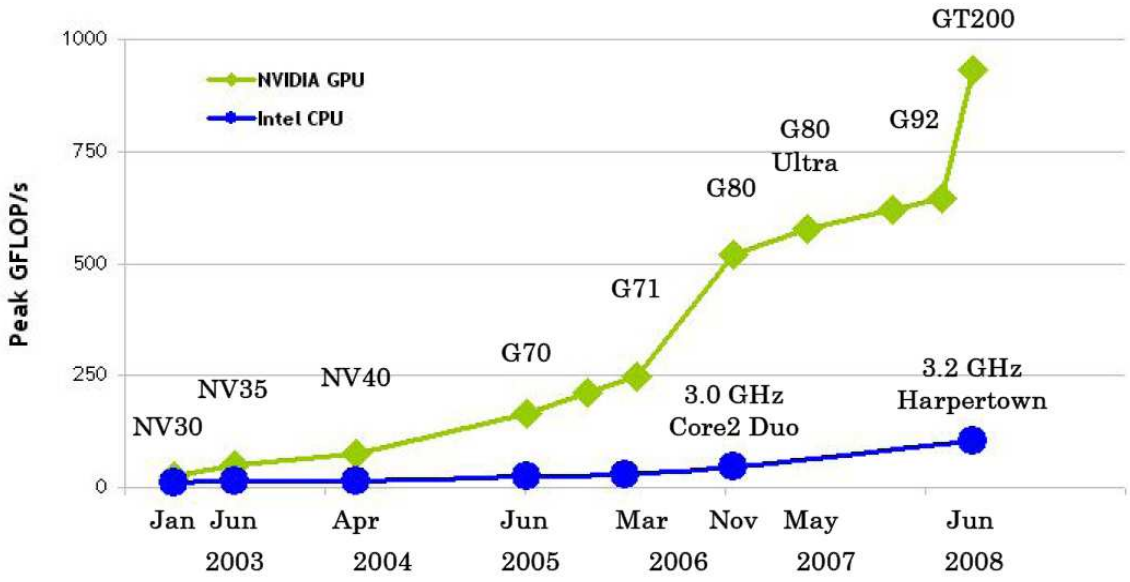
Giga, Terra, Peta, ...

Verdopplung der Rechenleistung alle 13.5 Monate.

Liste einiger berühmter Computer:

Name	Zeit	Prozessoren	Rechenoperationen/s
CDC 7600	1971	1	1.240.000
Cray 2	1987	4	1.700.000.000
ASCI Red	1997	7264	1.068.000.000.000
IBM Roadrunner	2008	122400	1.026.000.000.000.000

General Purpose GPU-Computing



Quelle: NVIDIA

Effiziente Algorithmen

- Schnelle Rechner sind nicht alles!
- Rechenzeiten zur (approximativen) Lösung spezieller linearer Gleichungssysteme:

N	Gauß ($\frac{2}{3}N^3$)	Mehrgitter ($100N$)
1.000	0.66 s	10^{-4} s
10.000	660 s	10^{-3} s
100.000	7.6 Tage	10^{-2} s
$1 \cdot 10^6$	21 Jahre	0.1 s
$1 \cdot 10^7$	21.000 Jahre	1 s

  (für einen Rechner mit 1 GFLOP/s)
- Parallelisierung kann einem ineffizienten Verfahren nicht helfen!
- $\Rightarrow$ Schnelle Rechner **und** schnelle Algorithmen

1.5 Inhalt der Vorlesung

Inhalt der Vorlesung

I Hardware
- Prozessorentwicklung, Pipelining, SIMD, MIMD, Caches, Cachekonsistenz, Multicore
- Verbindungsnetzwerke
- Grafikprozessoren

II Programmierung
- gemeinsamer Speicher: Locks, Semaphore
- PThreads, OpenMP
- verteilter Speicher: MPI-I, MPI-II
- RPC, Corba
- CUDA
- Illustration mittels N-Körper-Problem

III Algorithmen
- Bewertung von Algorithmen, prinzipielles Vorgehen, Lastverteilung
- dichtbesetzte Matrizen
- dünnbesetzte Gleichungssysteme
- Sortieren, N-Körper-Problem

Links

- Vorlesung `http://hal.iwr.uni-heidelberg.de/phlr.html`
- ASCI Program `https://asc.llnl.gov/`
- Seymour Cray `http://research.microsoft.com/users/gbell/craytalk/`
- TOP 500 Supercomputer Sites `http://www.top500.org/`

2 Rundgang

2.1 Parametrisierte Prozesse

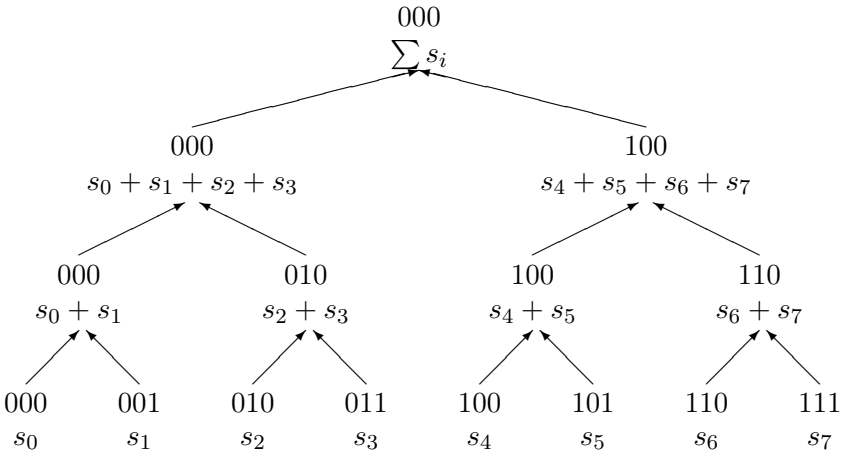
Parametrisieren von Prozessen

- Single Program Multiple Date (SPMD).
- Erlaubt Formulierung skalierbarer Algorithmen.

parallel many-process-scalar-product
{
    const int N; // Problemgröße
    const int P; // Anzahl Prozesse
    double $x[N]$, $y[N]$; // Vektoren
    double $s = 0$; // Resultat
    process Π [**int** $p \in \{0, ..., P-1\}$]
    {
        int i; **double** $ss = 0$;
        for $(i = N * p/P;\ i < N * (p+1)/P;\ i{+}{+})$
            ss += $x[i]$*$y[i]$;
        $[s = s + ss]$; // Hier warten dann doch wieder alle
    }
}

2.2 Parallele Summenberechnung

Parallele Summenberechnung



Implementierung Parallele Summe

```
parallel parallel-sum-scalar-product
{
    const int d = 4;
    const int N = 100;              // Problemgröße
    const int P = 2^d;              // Anzahl Prozesse
    double x[N], y[N];              // Vektoren
    double s[P] = {0[P]};           // Resultat
    int flag[P] = {0[P]};           // Prozess p ist fertig

    process Π [int p ∈ {0, ..., P − 1}]
    {
        int i, r, m, k;

        for (i = N ∗ p/P; i < N ∗ (p + 1)/P; i++)
            s[p]+ = x[i] ∗ y[i];

        for (i = 0; i < d; i++)
        {
            r = p & [∼ (Σ_{k=0}^{i} 2^k)];      // lösche letzten i + 1 bits
            m = r | 2^i;                        // setze Bit i
            if (p == m) flag[m]=1;
            if (p == r)
            {
                while (!flag[m]);               // Bedingungssynchronisation
                s[p] = s[p] + s[m];
            }
        }
    }
}
```

2.3 Lokalisieren

Lokalisieren der Vektoren

```
parallel local-data-scalar-product
{
    const int P, N;
    double s = 0;
```

```
    process Π [ int p ∈ {0,...,P−1}]
    {
        double x[N/P+1], y[N/P+1];
                                    // Lokaler Ausschnitt der Vektoren
        int i;
        double ss=0;

        for (i=0,i<(p+1)*N/P−p*N/P;i++) ss=ss+x[i]*y[i];
        [s=s+ss;]
    }
}
```

2.4 Nachrichtenaustausch

Summenberechnung mit Nachrichtenaustausch

parallel message-passing-scalar-product
{
    const int d, P= 2^d, N; // Konstanten!

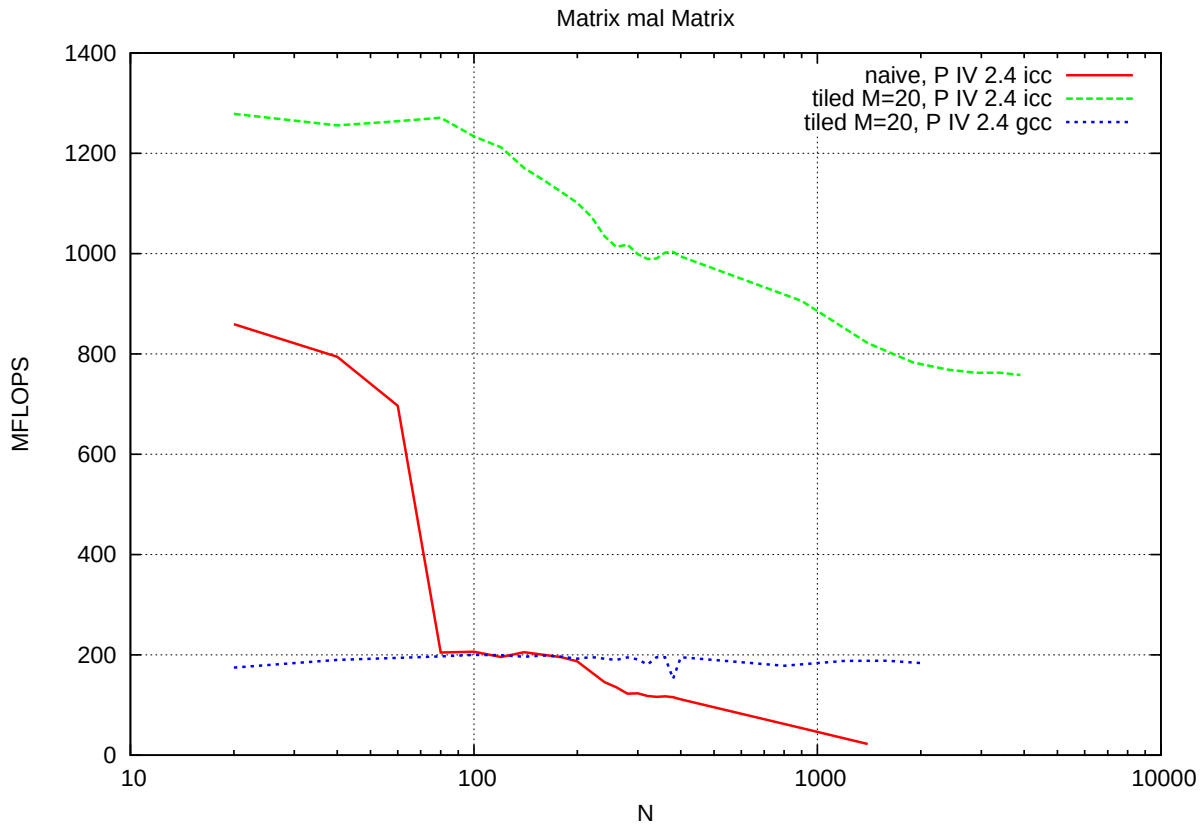
    process Π [**int** $p \in \{0,\ldots,P-1\}$]
    {
        double $x[N/P]$, $y[N/P]$;// Lokaler Ausschnitt der Vektoren
        int i, r, m;
        double $s=0, ss$;

        for ($i=0$,$i<(p+1)*N/P-p*N/P$;i++) $s=s+x[i]*y[i]$;
        for ($i=0$,$i<d$,i++) // d Schritte
        {
            $r = p \;\&\; \left[\sim \left(\sum_{k=0}^{i} 2^k\right)\right]$;
            $m = r \mid 2^i$;
            if ($p == m$)
                send(Π_r,s);
            if ($p == r$)
            {
                receive(Π_m,ss);
                $s = s + ss$;
            }
        }
    }
}

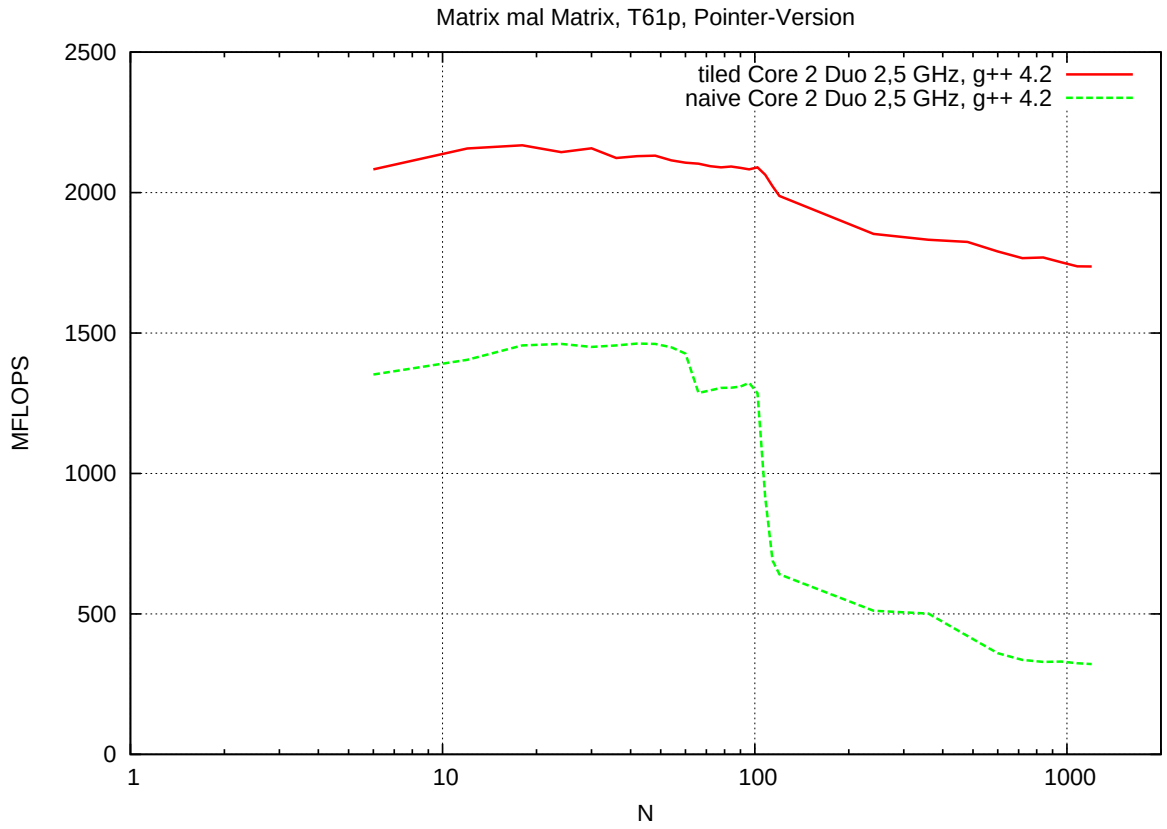
3 Sequentielle Rechnerarchitektur

3.1 Performance

Matrixmultiplikation, Pentium IV 2.4 GHz

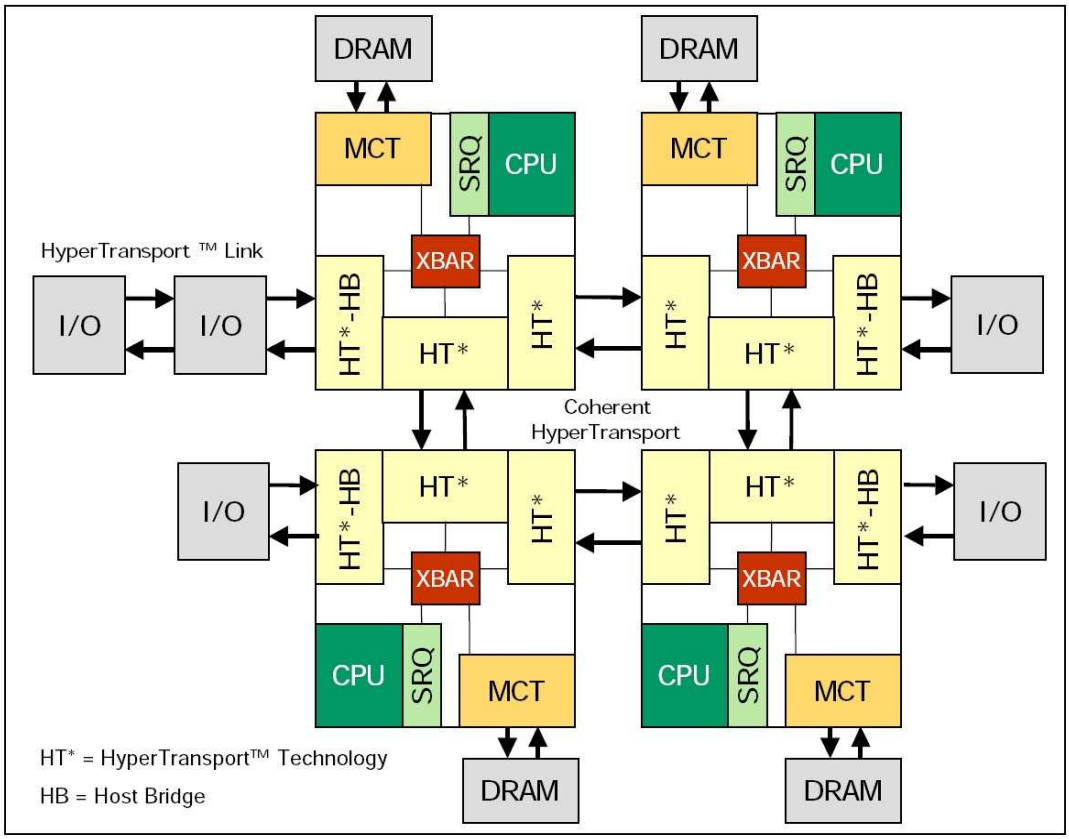


Matrixmultiplikation, Core 2 Duo 2.4 GHz

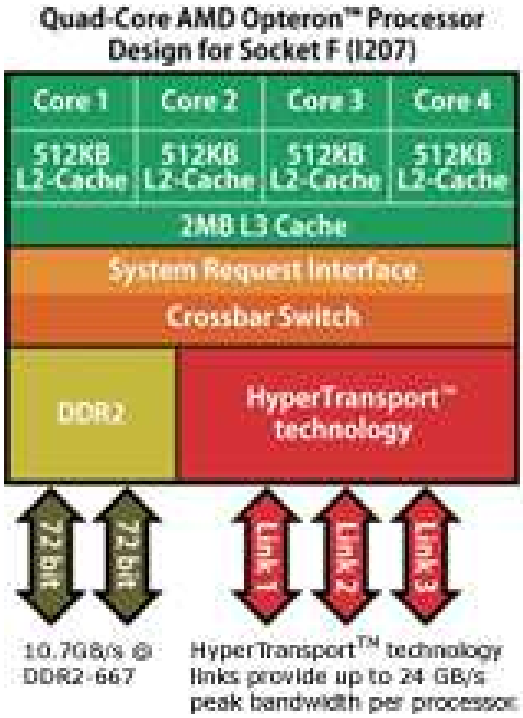


3.2 AMD Opteron Prozessor

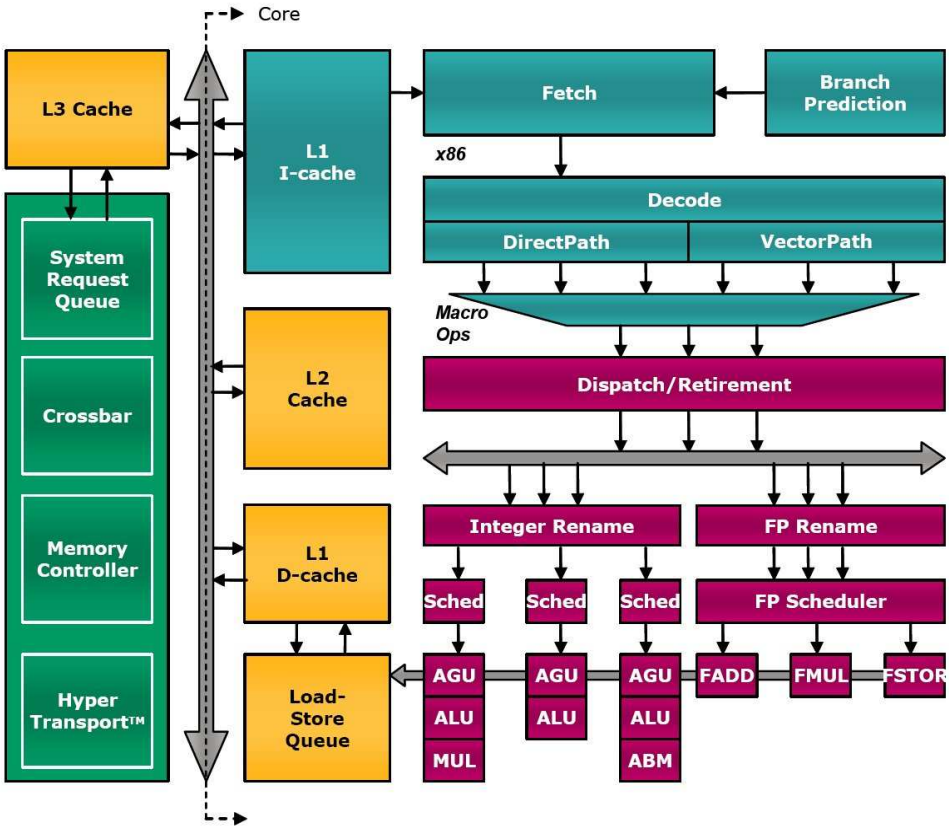
Hammer Architektur Generation 8, 2001 eingeführt



Barcelona QuadCore, Generation 10h, 2007



Barcelona Core



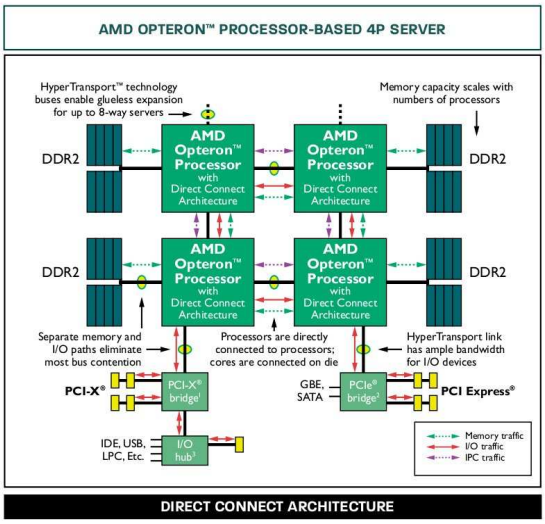
Barcelona Details

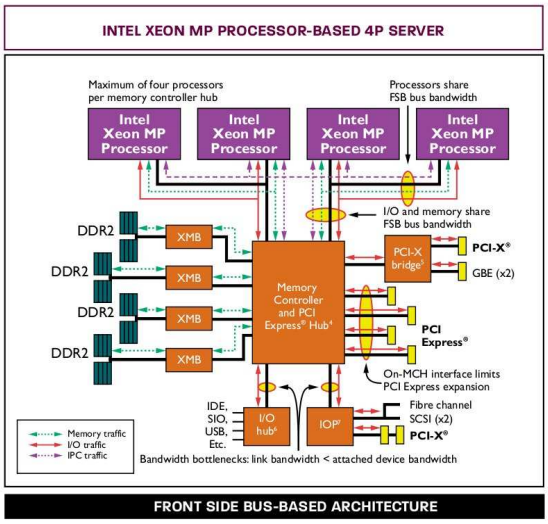
- L1 Cache
  - 64K Instruktionen, 64K Daten.
  - Cache line 64 Bytes.
  - 2 Wege assoziativ, write-allocate, writeback, least-recently-used.
  - MOESI.
- L2 Cache
  - Victim cache: enthält nur Blöcke, die aus dem L1 cache verdrängt wurden.
  - Größe implementierungsabhängig.
- L3 Cache
  - non-inclusive: Daten sind entweder im L1 *oder* L3.
  - Victim Cache, d.h. aus dem L2 verdrängte Blöcke landen hier.
  - Heuristische Sharing-Erkennung.
- Superskalar
  - Instruktionsdekodierung, Integer units, FP units, address generation jeweils dreifach (3 OPS pro Zyklus).

  - Out of order execution, branch prediction.
- Pipelining: 12 Stufen integer, 17 Stufen floating-point (Hammer).
- Integrierter Speichercontroller, 128 Bit breit.
- HyperTransport: Kohärenter Zugriff auf entfernten Speicher.

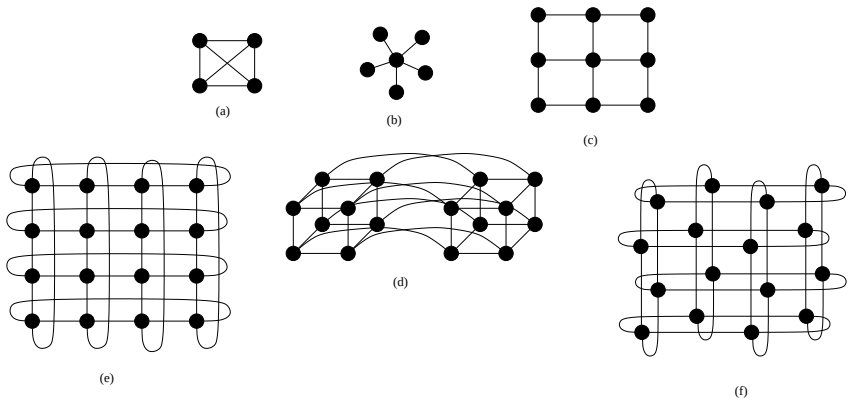
3.3 Kommunikationsnetzwerke

Opteron/Xeon Vergleich





Netzwerktopologien I



(a) full connected, (b) star, (c) array (d) hypercube, (e) torus, (f) folded torus

- *Hypercube*: der Dimension d hat 2^d Prozessoren. Prozessor p ist mit q verbunden wenn sich deren Binärdarstellungen *in genau einem Bit* unterscheiden.

- Netzwerkknoten: Früher (vor 1990) war das der Prozessor selbst, heute sind es dedizierte Kommunikationsprozessoren

Netzwerktopologien II

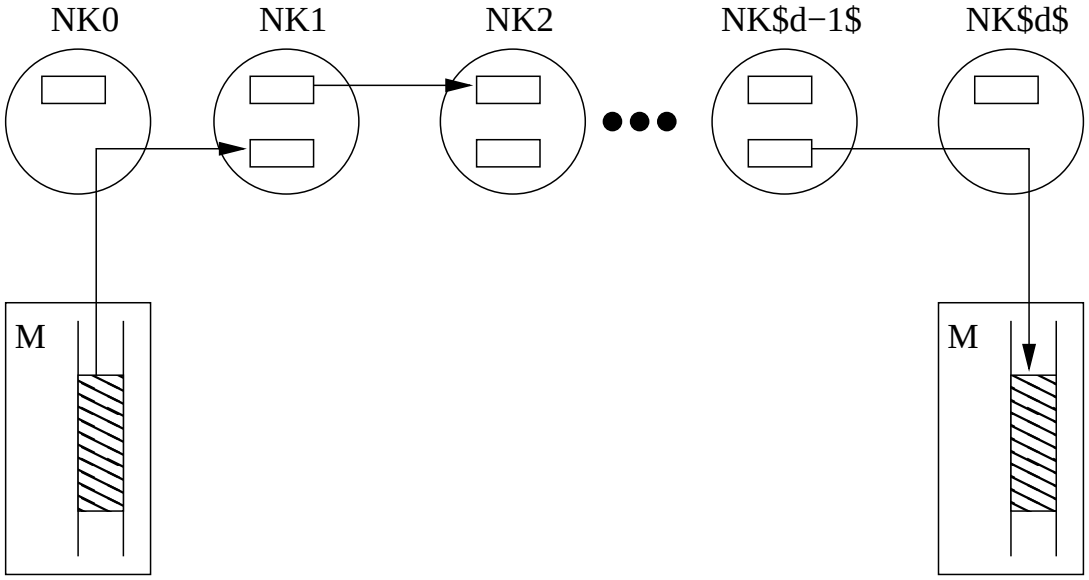
Kennzahlen:

Topologie	k	D	l	B	Sym
Volle Verb.	$P-1$	1	$\frac{P(P-1)}{2}$	$\left(\frac{P}{2}\right)^2$	ja
Hypercube	d	d	$d\frac{P}{2}$	$\frac{P}{2}$	ja
2D-Torus	4	$2\left\lfloor\frac{r}{2}\right\rfloor$	$2P$	$2r$	ja
2D-Feld	4	$2(r-1)$	$2P-2r$	r	nein
Ring	2	$\left\lfloor\frac{P}{2}\right\rfloor$	P	2	ja
Binärer Baum	3	$2(h-1)$	$P-1$	1	nein

$d = \log_2 P, P = r \times r, h = \lceil \log_2 P \rceil$

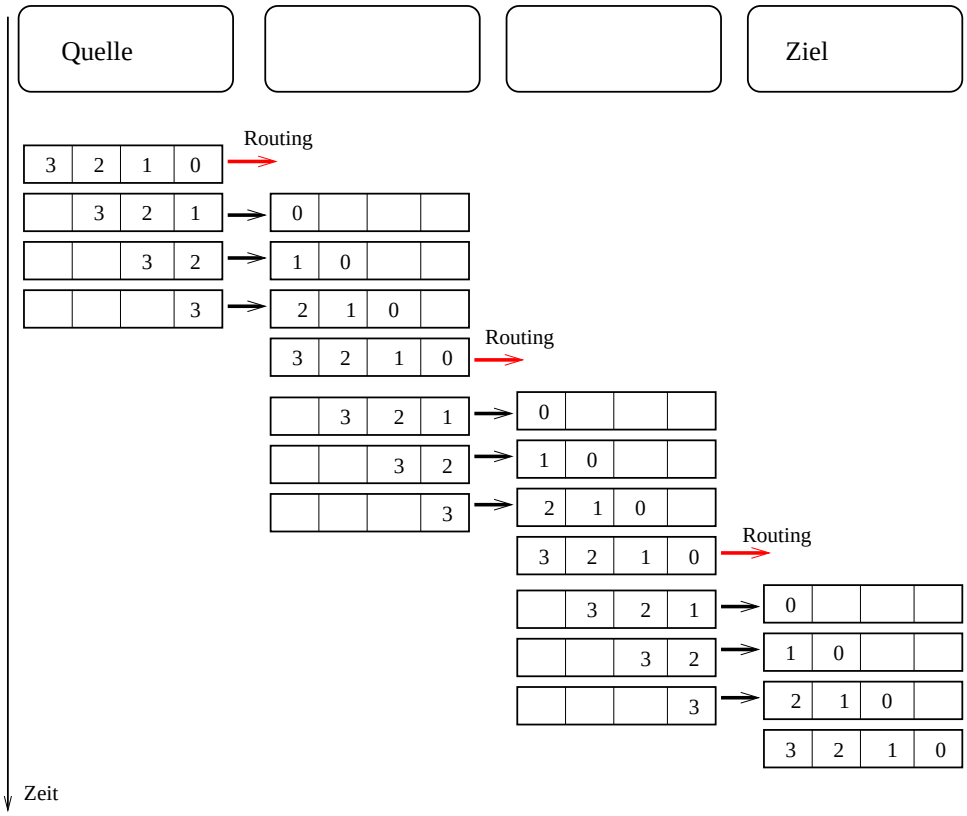
Store & Forward Routing

Store-and-forward routing: Nachricht der Länge n wird in Pakete der Länge N zerlegt. Pipelining auf Paketebene: Paket wird aber vollständig im NK gespeichert



Store & Forward Routing

Übertragung eines Paketes:



Store & Forward Routing

Laufzeit:

$$\begin{aligned} t_{SF}(n,N,d) &= t_s + d(t_h + Nt_b) + \left(\frac{n}{N} - 1\right)(t_h + Nt_b) \\ &= t_s + t_h\left(d + \frac{n}{N} - 1\right) + t_b\left(n + N(d-1)\right). \end{aligned}$$

t_s: Zeit, die auf Quell– und Zielrechner vergeht bis das Netzwerk mit der Nachrichenübertragung beauftragt wird, bzw. bis der empfangende Prozess benachrichtigt wird. Dies ist der Softwareanteil des Protokolls.

t_h: Zeit die benötigt wird um das erste Byte einer Nachricht von einem Netzwerkknoten zum anderen zu übertragen (*engl.* node latency, hop–time).

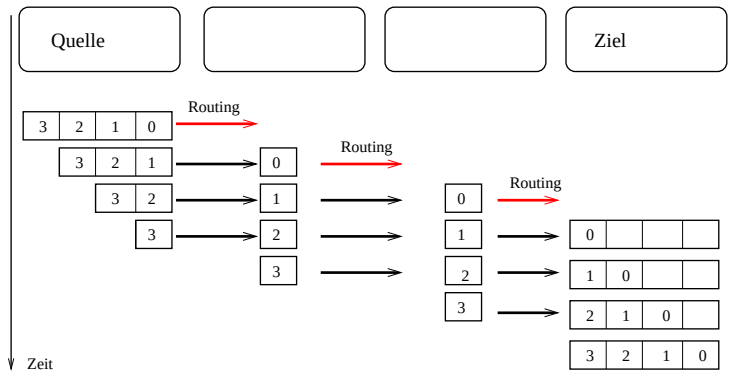
t_b: Zeit für die Übertragung eines Byte von Netzwerkknoten zu Netzwerkknoten.

d: Hops bis zum Ziel.

Cut-Through Routing

Cut-through routing oder *wormhole routing*: Pakete werden nicht zwischengespeichert, jedes Wort (sog. *flit*) wird sofort an nächsten Netzwerkknoten weitergeleitet

Übertragung eines Paketes:



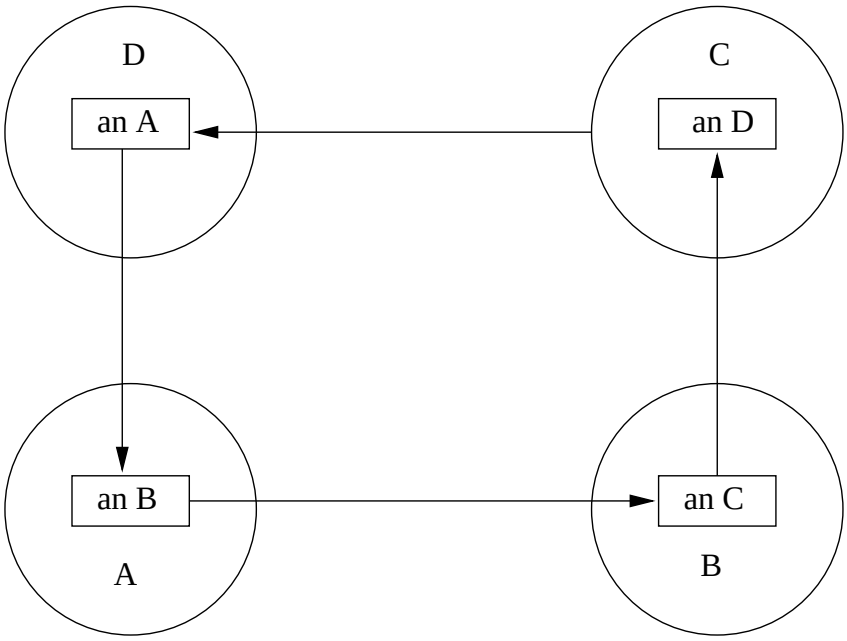
Laufzeit:

$$t_{CT}(n, N, d) = t_s + t_h d + t_b n$$

Zeit für kurze Nachricht ($n = N$): $t_{CT} = t_s + dt_h + Nt_b$. Wegen $dt_h \ll t_s$ (Hardware!) quasi entfernungsunabhängig

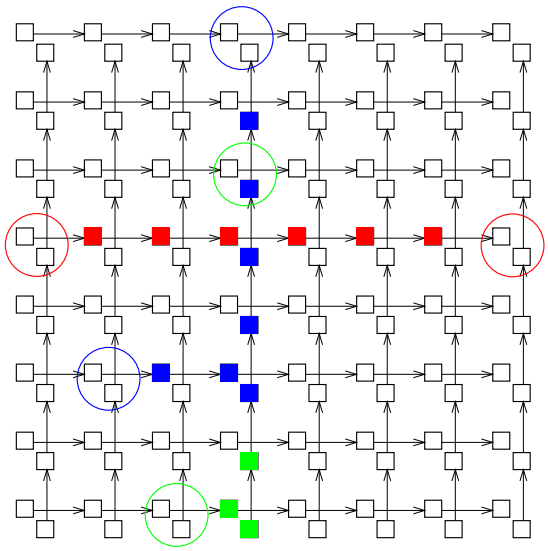
Deadlock

In paketvermittelnden Netzwerken besteht die Gefahr des *store-and-forward deadlock*:



Deadlock

Zusammen mit cut-through routing:



Verklemmungsfreies „dimension routing“. Beispiel 2D-Gitter: Zerlege Netzwerk in $+x$, $-x$, $+y$ und $-y$ Netzwerke mit jeweils eigenen Puffern. Nachricht läuft erst in Zeile, dann in Spalte.

4 Synchronisationsmechanismen bei gemeinsamen Speicher

4.1 Hardware Locks

Hardware-OPs

- Verschiedene *Lock* Algorithmen erfordern Speicherkonsistenz.
- Hoher extra Aufwand erforderlich, um Speicherkonsistenz zu garantieren.
- Verfahren sind nicht portabel und erfordern genaue Kenntnis der Architektur.
- Spezielle Hardware Operationen vereinfachen die Konstruktion von *Locks* und garantieren die erforderliche Speicherkonsistenz.

Übersicht der Hardware-OPs

Klassische OPs

- **test-and-set**: (*atomar*) lese Speicherstelle und schreibe 1 rein.
- **atomic-swap**: (*atomar*) tausche Speicherstelle mit Register.
- **fetch-and-increment**: (*atomar*) lese Speicherstelle und schreibe *Wert*+1 zurück.

Andere OPs

- **compare-and-swap** (*atomar*) vertausche
- **load-linked** / **store-conditional** (*atomar*) lese

test-and-set / TAS-Lock

Lock baut auf dem atomaren Befehl *test-and-set* auf.

```
bool test-and-set(bool * target)
{
  bool state = *target;
  *target = true;
  return state;
}

parallel TAS-Lock
{
  bool lock = false
  process Π [int p ∈ {0,...,P − 1}] {
    while (1) {
      while test-and-set(&lock); %\only<3>{\large\structure{← cache misses}
      %\textit{kritischer Abschnitt};%
      lock = false;
      %\textit{unkritischer Abschnitt};%
```

```
    }
  }
}
```

TTAS-Lock

Verbesserung des TAS-Lock, welche die cache misses reduziert.

```
parallel TTAS-Lock // test and test-and-set
{
  bool lock = false
  process Π [int p ∈ {0,...,P-1}] {
    while (1) {
      while (1) {
        if (lock == false)
          if (test-and-set(&lock))
            break;
      }
      %\textit{kritischer Abschnitt};%
      lock = false;
      %\textit{unkritischer Abschnitt};%
    }
  }
}
```

Exponential Backoff

- Verbesserung des TTAS-Lock. Der erneute Versuch das Lock zu bekommen wird etwas verzögrt.
- Die Verzögerung wurde zufällig gewählt.
- Die obere Zufallsgrenze verdoppelt sich nach jedem weiteren Fehlversuch.

```
parallel ExpBackoff-Lock
{
  bool lock = false;
  process Π [int p ∈ {0,...,P-1}] {
    while (1) {
      while (1) {
        if (lock == false)
          if (test-and-set(&lock))
            break;
          else {
            delay ∈ [mindelay,maxdelay];
            sleep(delay);
```

```
            maxdelay*=2;
          }
        }
        %\textit{kritischer Abschnitt};%
        lock = false;
        %\textit{unkritischer Abschnitt};%
      }
    }
  }
```

atomic-swap

- tauscht den Inhalt einer Speicherstelle und eines Registers.
- *atomic-swap* kann mehr als *test-and-set*.

```
TYPE atomic-swap(TYPE * target, TYPE value)
{
  TYPE tmp = *target;
  *target = value;
  return tmp;
}
```

test-and-set mit Hilfe von *atomic-swap*:

```
bool test-and-set(bool * target) {
  return atomic-swap(target, true);
}
```

fetch-and-increment

```
TYPE fetch-and-increment(TYPE * target)
{
  TYPE value = *target;
  *target = value + 1;
  return value;
}
```

Lock basierend auf *fetch-and-increment*

```
parallel FAI-Lock // naiv implementation
{
  int ticketnumber, turn;
  process Π [int p ∈ {0,...,P − 1}] {
    while (1) {
      int myturn = fetch-and-increment(ticketnumber);
      while (turn != myturn);
```

```
        %\textit{kritischer Abschnitt};%
        fetch-and-increment(turn);
        %\textit{unkritischer Abschnitt};%
      }
    }
  }
```

compare-and-swap

Wird in Prozessoren von Intel, AMD und Sun verwendet.

```
bool compare-and-swap(TYPE * target,
                      TYPE expect, TYPE value)
{
  TYPE tmp = *target;
  if (tmp == expect) {
    *target = value;
    return true;
  }
  else {
    return false;
  }
}
```

load-linked / store-conditional

Wird in echten RISC Prozessoren verwendet, z.B. ARM, Alpha, MIPS, PowerPC.

- *LL* liest von einer Speicheradresse.
- Ein späteres *SC* versucht einen neuen Wert in dieser Adresse zu speichern.
- *SC* ist erfolgreich, wenn der Wert der Adresse seit dem *LL* nicht verändert wurde.
- Wenn der Wert an der Adresse zwischenzeitlich geändert wurde, selbst wenn er jetzt wieder den ursprünglichen Wert hat, schlägt *SC* fehl.

4.2 Barrieren

Erster Entwurf einer Barriere

parallel barrier-1
{
  const int P=8;
  int *count*=0;

```
    int release=0;

    process Π [int p ∈ {0, ..., P − 1}]
    {
        while (1)
        {
            Berechnung;
            CSenter;                    // Eintritt
            if (count==0) release=0;    // Zurücksetzen
            count=count+1;              // Zähler erhöhen
            CSexit;                     // Verlassen
            if (count==P) {
                count=0;                // letzter löscht
                release=1;              // und gibt frei
            } else
                while (release==0) ;    // warten
        }
    }
}
```

Barriere mit Richtungsumkehr

```
parallel sense-reversing-barrier
{
    const int P=8;
    int count=0;
    int release=0;

    process Π [int p ∈ {0, ..., P − 1}]
    {
        int local_sense = release;
        while (1)
        {
            Berechnung;
            local_sense = 1-local_sense; // Richtung wechseln
            CSenter;                    // Eintritt
            count=count+1;              // Zähler erhöhen
            CSexit;                     // Verlassen
            if (count==P) {
                count=0;                // letzter löscht
                release=local_sense;    // und gibt frei
            } else
                while (release≠local_sense) ;
```

```
        }
    }
}
```

Barriere mit Baum

```
parallel tree-barrier
{
    const int d = 4;
    const int P = 2^d;
    int arrived[P]={0,...,0};
    int continue[P]={0,...,0};

    process Π [int p ∈ {0,...,P−1}]
    {
        int i, r, m, k;

        while (1) {
            Berechnung;
            for (i = 0; i < d; i++)                    // aufwärts
            {
                r = p & [∼ (Σ_{k=0}^{i} 2^k)];         // Bits 0 bis i löschen
                m = r | 2^i;                           // Bit i setzen
                if (p == m) arrived[m]=1;
                if (p == r)
                {
                    while(¬arrived[m]) ;               // warte
                    arrived[m]=0;
                }
            }
            for (i = d − 1; i ≥ 0; i–)                 // abwärts
            {
                r = p & [∼ (Σ_{k=0}^{i} 2^k)];         // Bits 0 bis i löschen
                m = r | 2^i;
                if (p == m)
                {
                    while(¬continue[m]) ;
                    continue[m]=0;
                }
                if (p == r) continue[m]=1;
            }
```

```
        }
    }
}
```

Barriere mit rekursiver Verdopplung

```
parallel recursive-doubling-barrier
{
    const int d = 4;
    const int P = 2^d;
    int arrived[d][P]={0[P · d]};

    process Π [int p ∈ {0, ..., P − 1}]
    {
        int i, q;

        while (1) {
            Berechnung;
            for (i = 0; i < d; i++)              // alle Stufen
            {
                q = p ⊕ 2^i;                     // Bit i umschalten
                while (arrived[i][p]) ;
                arrived[i][p]=1;
                while (¬arrived[i][q]) ;
                arrived[i][q]=0;
            }
        }
    }
}
```

4.3 Semaphore

$m/n/1$ Erzeuger-Verbraucher-Problem

```
parallel prod-con-nm1
{
    const int m = 3, n = 5;
    Semaphore empty=1;          // freier Pufferplatz
    Semaphore full=0;           // abgelegter Auftrag
    T buf;                      // der Puffer

    process P [int i ∈ {0, ..., m − 1}]
```

```
    {
        while (1)
        {
            Erzeuge Auftrag t;
            P(empty);                  // Ist Puffer frei?
            buf = t;                   // speichere Auftrag
            V(full);                   // Auftrag abgelegt
        }
    }

    process C [int j ∈ {0, ..., n − 1}]
    {
        while (1)
        {
            P(full);                   // Ist Auftrag da?
            t = buf;                   // entferne Auftrag
            V(empty);                  // Puffer ist frei
            Bearbeite Auftrag t;
        }
    }
}
```

1/1/k Erzeuger-Verbraucher-Problem

```
parallel prod-con-11k
{
    const int k = 20;
    Semaphore empty=k;                 // zählt freie Pufferplätze
    Semaphore full=0;                  // zählt abgelegte Aufträge
    T buf[k];                          // der Puffer
    int front=0;                       // neuester Auftrag
    int rear=0;                        // ältester Auftrag

    process P
    {
        while (1)
        {
            Erzeuge Auftrag t;
            P(empty);                  // Ist Puffer frei?
            buf[front] = t;            // speichere Auftrag
            front = (front+1) mod k;   // nächster freier Platz
            V(full);                   // Auftrag abgelegt
        }
```

```
    }

    process C
    {
        while (1)
        {
            P(full);                    // Ist Auftrag da?
            t = buf[rear];              // entferne Auftrag
            rear = (rear+1) mod k;      // nächster Auftrag
            V(empty);                   // Puffer ist frei
            Bearbeite Auftrag t;
        }
    }
}
```

$m/n/k$ Erzeuger-Verbraucher-Problem

```
parallel prod-con-mnk
{
    const int k = 20, m = 3, n = 6;
    Semaphore empty=k;          // zählt freie Pufferplätze
    Semaphore full=0;           // zählt abgelegte Aufträge
    T buf[k];                   // der Puffer
    int front=0;                // neuester Auftrag
    int rear=0;                 // ältester Auftrag
    Semaphore mutexP=1;         // Zugriff der Erzeuger
    Semaphore mutexC=1;         // Zugriff der Verbraucher

    process P [int i ∈ {0, ..., m − 1}]
    {
        while (1)
        {
            Erzeuge Auftrag t;
            P(empty);           // Ist Puffer frei?
            P(mutexP);          // manipuliere Puffer
            buf[front] = t;     // speichere Auftrag
            front = (front+1) mod k; // nächster freier Platz
            V(mutexP);          // fertig mit Puffer
            V(full);            // Auftrag abgelegt
        }
    }

    process C [int j ∈ {0, ..., n − 1}]
```

```
    {
        while (1)
        {
            P(full);                    // Ist Auftrag da?
            P(mutexC);                  // manipuliere Puffer
            t = buf[rear];              // entferne Auftrag
            rear = (rear+1) mod k;      // nächster Auftrag
            V(mutexC);                  // fertig mit Puffer
            V(empty);                   // Puffer ist frei
            Bearbeite Auftrag t;
        }
    }
}
```

4.4 Philosophenproblem

Die speisenden Philosophen: Naive Lösung

```
parallel philosphers–1
{
    const int P = 5;                    // Anzahl Philosophen
    Semaphore forks[P] = { 1 [P] };     // Gabeln

    process Philosopher [int p ∈ {0, ..., P − 1}]
    {
        while (1)
        {
            Denke;
            P(fork[p]);                 // rechte Gabel
            P(fork[(p + 1) mod P]);     // linke Gabel
            Esse;
            V(fork[p]);                 // rechte Gabel
            V(fork[(p + 1) mod P]);     // linke Gabel
        }
    }
}
```

Die speisenden Philosophen: Faire Lösung

```
parallel philosphers–2
{
    const int P = 5;                    // Anzahl Philosophen
```

```
    const int think=0, hungry=1, eat=2;
    Semaphore mutex=1;
    Semaphore s[P] = { 0 [P] };        // essender Philosoph
    int state[P] = { think [P] };      // Zustand

    process Philosopher [int p ∈ {0, ..., P − 1}]
    {
        void test (int i)
        {
            int r=(i + P − 1) mod P, l=(i + 1) mod P;

            if (state[i]==hungry ∧ state[l]≠eat ∧ state[r]≠eat)
            {
                state[i] = eat;
                V(s[i]);
            }
        }

        while (1)
        {
            Denke;

            P(mutex);                  // Gabeln nehmen
            state[p] = hungry;
            test(p);
            V(mutex);
            P(s[p]);

            Esse;

            P(mutex);                  // Gabeln weglegen
            state[p] = think;
            test((p + P − 1) mod P);
            test((p + 1) mod P);
            V(mutex);
        }
    }
}
```

4.5 Leser-Schreiber-Problem

Leser-Schreiber-Problem: Naive Lösung

```
parallel readers–writers–1
{
    const int m = 8, n = 4;              // Anzahl Leser und Schreiber
    Semaphore rw=1;                      // Zugriff auf Datenbank
    Semaphore mutexR=1;                  // Anzahl Leser absichern
    int nr=0;                            // Anzahl zugreifender Leser

    process Reader [int i ∈ {0, ..., m − 1}]
    {
        while (1)
        {
            P(mutexR);                   // Zugriff Leserzähler
            nr = nr+1;                   // Ein Leser mehr
            if (nr==1) P(rw);            // Erster wartet auf DB
            V(mutexR);                   // nächster Leser kann rein

            lese Datenbank;

            P(mutexR);                   // Zugriff Leserzähler
            nr = nr-1;                   // Ein Leser weniger
            if (nr==0) V(rw);            // Letzter gibt DB frei
            V(mutexR);                   // nächster Leser kann rein
        }
    }
    process Writer [int j ∈ {0, ..., n − 1}]
    {
        while (1)
        {
            P(rw);                       // Zugriff auf DB
            schreibe Datenbank;
            V(rw);                       // gebe DB frei
        }
    }
}
```

Leser-Schreiber-Problem: Faire Lösung

```
parallel readers–writers–2
{
    const int m = 8, n = 4;              // Anzahl Leser und Schreiber
    int nr=0, nw=0, dr=0, dw=0;          // Zustand
    Semaphore e=1;                       // Zugriff auf Warteschlange
    Semaphore r=0;                       // Verzögern der Leser
```

```
Semaphore w=0;                  // Verzögern der Schreiber
const int reader=1, writer=2;   // Marken
int buf[n + m];                 // Wer wartet?
int front=0, rear=0;            // Zeiger

int wake_up (void)              // darf genau einer ausführen
{
    if (nw==0 ∧ dr>0 ∧ buf[rear]==reader){
        dr = dr-1;
        rear = (rear+1) mod (n + m);
        V(r);
        return 1;               // habe einen Leser geweckt
    }
    if (nw==0 ∧ nr==0 ∧ dw>0 ∧ buf[rear]==writer){
        dw = dw-1;
        rear = (rear+1) mod (n + m);
        V(w);
        return 1;               // habe einen Schreiber geweckt
    }
    return 0;                   // habe keinen geweckt
}

process Reader [int i ∈ {0, ..., m − 1}]{
    while (1) {
        P(e);                   // will Zustand verändern
        if(nw>0 ∨ dw>0){
            buf[front] = reader;    // in Warteschlange
            front = (front+1) mod (n + m);
            dr = dr+1;
            V(e);               // Zustand freigeben
            P(r);               // warte bis Leser dran sind
                                // hier ist e = 0 !
        }
        nr = nr+1;              // hier ist nur einer
        if (wake_up()==0)       // kann einer geweckt werden?
            V(e);               // nein, setze e = 1

        lese Datenbank;

        P(e);                   // will Zustand verändern
        nr = nr-1;
        if (wake_up()==0)       // kann einer geweckt werden?
            V(e);               // nein, setze e = 1
    }
}
```

```
    process Writer [int j ∈ {0, ..., n − 1}]
    {
        while (1){
            P(e);                          // will Zustand verändern
            if(nr>0 ∨ nw>0){
                buf[front] = writer;       // in Warteschlange
                front = (front+1) mod (n + m);
                dw = dw+1;
                V(e);                      // Zustand freigeben
                P(w);                      // warte bis an der Reihe
                                           // hier ist e = 0 !
            }
            nw = nw+1;                     // hier ist nur einer
            V(e);                          // hier braucht keiner geweckt werden

            schreibe Datenbank;            // exklusiver Zugriff

            P(e);                          // will Zustand verändern
            nw = nw-1;
            if (wake_up()==0)              // kann einer geweckt werden?
                V(e);                      // nein, setze e = 1
        }
    }
}
```

5 PThreads

5.1 Allgemeines

Prozesse und Threads

Ein Unix-Prozess hat

- IDs (process,user,group)
- Umgebungsvariablen
- Verzeichnis
- Programmcode
- Register, Stack, Heap
- Dateideskriptoren, Signale
- message queues, pipes, shared memory Segmente
- Shared libraries

Jeder Prozess besitzt seinen eigenen Adressraum

Threads existieren innerhalb eines Prozesses

Threads teilen sich einen Adressraum

Ein Thread besteht aus

- ID
- Stack pointer
- Register
- Scheduling Eigenschaften
- Signale

Erzeugungs- und Umschaltzeiten sind kürzer

„Parallele Funktion“

Pthreads

- Jeder Hersteller hatte eine eigene Implementierung von Threads oder „light weight processes“
- 1995: IEEE POSIX 1003.1c Standard (es gibt mehrere “drafts“)
- Standard Dokument ist kostenpflichtig
- Definiert Threads in portabler Weise
- Besteht aus C Datentypen und Funktionen
- Header file `pthread.h`
- Bibliotheksname nicht genormt. In Linux `-lpthread`
- Übersetzen in Linux: `gcc <file> -lpthread`

Pthreads Übersicht

Alle Namen beginnen mit `pthread_`

- `pthread_` Thread Verwaltung und sonstige Routinen
- `pthread_attr_` Thread Attributobjekte
- `pthread_mutex_` Alles was mit Mutexvariablen zu tun hat
- `pthread_mutex_attr_` Attribute für Mutexvariablen
- `pthread_cond_` Bedingungsvariablen (condition variables)
- `pthread_cond_attr_` Attribute für Bedingungsvariablen

5.2 Thread Management

Erzeugen von Threads

- `pthread_t` : Datentyp für einen Thread.
- Opaquer Typ: Datentyp wird in der Bibliothek definiert und wird von deren Funktionen bearbeitet. Inhalt ist implementierungsabhängig.
- `int pthread_create(thread,attr,start_routine,arg)` : Startet die Funktion `start_routine` als Thread.
  - `thread` : Zeiger auf eine `pthread_t` Struktur. Dient zum identifizieren des Threads.
  - `attr` : Threadattribute besprechen wir unten. Default ist `NULL`.
  - `start_routine` Zeiger auf eine Funktion vom Typ `void* func (void*);`
  - `arg` : `void*`-Zeiger der der Funktion als Argument mitgegeben wird.
  - Rückgabewert größer Null zeigt Fehler an.
- Threads können weitere Threads starten, maximale Zahl von Threads ist implementierungsabhängig

Beenden von Threads

- Es gibt folgende Möglichkeiten einen Thread zu beenden:
  - Der Thread beendet seine `start_routine()`
  - Der Thread ruft `pthread_exit()`
  - Der Thread wird von einem anderen Thread mittels `pthread_cancel()` beendet
  - Der Prozess wird durch `exit()` oder durch das Ender der `main()`-Funktion beendet
- `pthread_exit(void* status)`
  - Beendet den rufenden Thread. Zeiger wird gespeichert und kann mit `pthread_join` (s.u.) abgefragt werden (Rückgabe von Ergebnissen).
  - Falls `main()` diese Routine ruft so laufen existierende Threads weiter und der Prozess wird nicht beendet.
  - Schließt keine geöffneten Dateien!

Warten auf Threads

- Peer Modell: Mehrere gleichberechtigte Threads bearbeiten eine Aufgabe. Programm wird beendet wenn alle Threads fertig sind
- Erfordert Warten eines Threads bis alle anderen beendet sind
- Dies ist eine Form der Synchronisation
- `int pthread_join(pthread_t thread, void **status);`
  - Wartet bis der angegebene Thread sich beendet
  - Der Thread kann mittel `pthread_exit()` einen `void*`-Zeiger zurückgeben,
  - Gibt man `NULL` als Statusparameter, so verzichtet man auf den Rückgabewert

Thread Management Beispiel

```
#include <pthread.h>     /* for threads     */

void* prod (int *i) {    /* Producer thread */
  int count=0;
  while (count<100000) count++;
}

void* con (int *j) { /* Consumer thread */
  int count=0;
  while (count<1000000) count++;
}

int main (int argc, char *argv[]) { /* main program */
  pthread_t thread_p, thread_c; int i,j;

  i = 1;
  pthread_create(&thread_p,NULL,(void*(*)(void*)) prod,(void *) &i);
  j = 1;
  pthread_create(&thread_c, NULL,(void*(*)(void*)) con, (void *) &j);

  pthread_join(thread_p, NULL); pthread_join(thread_c, NULL);
  return(0);
}
```

Übergeben von Argumenten

- Übergeben von mehreren Argumenten erfordert Definition eines eigenen Datentyps:

  ```
  struct argtype {int rank; int a,b; double c;};
  struct argtype args[P];
  pthread_t threads[P];

  for (i=0; i<P; i++) {
    args[i].rank=i; args[i].a=...
      pthread_create(threads+i,NULL,(void*(*)(void*)) prod,
                     (void *)args+i);
  }
  ```

- Folgendes Beispiel enthält zwei Fehler:

  ```
  pthread_t threads[P];
  for (i=0; i<P; i++) {
    pthread_create(threads+i,NULL,(void*(*)(void*)) prod,&i);
  ```

  - Inhalt von `i` ist möglicherweise verändert bevor Thread liest
  - Falls `i` eine Stackvariable ist exisitiert diese möglicherweise nicht mehr

Thread Identifiers

- `pthread_t pthread_self(void);` Liefert die eigene Thread-ID

- `int pthread_equal(pthread_t t1, pthread_t t2);` Liefert wahr (Wert>0) falls die zwei IDs identisch sind
- Konzept des „opaque data type“

Join/Detach

- Ein Thread im Zustand `PTHREAD_CREATE_JOINABLE` gibt seine Resourcen erst frei, wenn `pthread_join` ausgeführt wird.
- Ein Thread im Zustand `PTHREAD_CREATE_DETACHED` gibt seine Resourcen frei sobald er beendet wird. In diesem Fall ist `pthread_join` nicht erlaubt.
- Default ist `PTHREAD_CREATE_JOINABLE`, das implementieren aber nicht alle Bibliotheken.
- Deshalb besser:

```
pthread_attr_t attr;
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
int rc = pthread_create(&t,&attr,(void*(*)(void*))func,NULL);
....
pthread_join(&t,NULL);
pthread_attr_destroy(&attr);
```

- Gibt Beispiel für die Verwendung von Attributen

5.3 Synchronisation

Mutex Variablen

- Mutex Variablen realisieren den wechselseitigen Ausschluss innerhalb der Pthreads Bibliothek
- Erzeugen und initialisieren einer Mutex Variable

```
pthread_mutex_t mutex;
pthread_mutex_init(&mutex,NULL);
```

  Mutex Variable ist nach Initialisierung im Zustand frei
- Eintritt in den kritischen Abschnitt:

```
pthread_mutex_lock(&mutex);
```

  Diese Funktion blockiert solange bis man drin ist.
- Verlasse kritischen Abschnitt

```
pthread_mutex_unlock(&mutex);
```

- Gebe Resourcen der Mutex Variable wieder frei

```
pthread_mutex_destroy(&mutex);
```

Bedingungsvariablen

- Bedingungsvariablen erlauben das *inaktive* Warten eines Prozesses bis eine gewisse Bedingung eingetreten ist
- Einfachstes Beispiel: Flaggenvariablen (siehe Beispiel unten)
- Zu einer Bedingungssynchronisation gehören *drei* Dinge:
  - Eine Variable vom Typ `pthread_cond_t`, die das inaktive Warten realisiert
  - Eine Variable vom Typ `pthread_mutex_t`, die den wechselseitigen Ausschluss beim Ändern der Bedingung realisiert
  - Eine globale Variable, deren Wert die Berechnung der Bedingung erlaubt

Erzeugen/Löschen

- `int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *attr);` initialisiert eine Bedingungsvariable

  Im einfachsten Fall: `pthread_cond_init(&cond,NULL)`
- `int pthread_cond_destroy(pthread_cond_t *cond);` gibt die Resourcen einer Bedingungsvariablen wieder frei

Wait

- `int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);` blockiert den aufrufenden Thread bis für die Bedingungsvariable die Funktion `pthread_signal()` aufgerufen wird
- Beim Aufruf von `pthread_wait()` muss der Thread auch im Besitz des Locks sein
- `pthread_wait()` verlässt das Lock und wartet auf das Signal in atomarer Weise
- Nach der Rückkehr aus `pthread_wait()` ist der Thread wieder im Besitz des Locks
- Nach Rückkehr muss die Bedingung nicht unbeding erfüllt sein
- Mit einer Bedingungsvariablen sollte man nur genau ein Lock verwenden

Signal

- `int pthread_cond_signal(pthread_cond_t *cond);` Weckt einen Prozess der auf der Bedingungsvariablen ein `pthread_wait()` ausgeführt hat. Falls keiner wartet hat die Funktion keinen Effekt.
- Beim Aufruf sollte der Prozess im Besitz des zugehörigen Locks sein
- Nach dem Aufruf sollte das Lock freigegeben werden. Erst die Freigabe des Locks erlaubt es dem wartenden Prozess aus der `pthread_wait()` Funktion zurückzukehren
- `int pthread_cond_broadcast(pthread_cond_t *cond);` weckt *alle* Threads die auf der Bedingungsvariablen ein `pthread_wait()` ausgeführt haben. Diese bewerben sich dann um das Lock.

Beispiel I

```
#include<stdio.h>
#include<pthread.h>     /* for threads     */

int arrived_flag=0,continue_flag=0;
pthread_mutex_t arrived_mutex, continue_mutex;
pthread_cond_t arrived_cond, continue_cond;

pthread_attr_t attr;

int main (int argc, char *argv[])
{
  pthread_t thread_p, thread_c;

  pthread_mutex_init(&arrived_mutex,NULL);
  pthread_cond_init(&arrived_cond,NULL);
  pthread_mutex_init(&continue_mutex,NULL);
  pthread_cond_init(&continue_cond,NULL);
```

Beispiel II

```
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

  pthread_create(&thread_p,&attr,(void*(*)(void*)) prod,NULL);
  pthread_create(&thread_c,&attr,(void*(*)(void*)) con ,NULL);

  pthread_join(thread_p, NULL);
  pthread_join(thread_c, NULL);

  pthread_attr_destroy(&attr);

  pthread_cond_destroy(&arrived_cond);
  pthread_mutex_destroy(&arrived_mutex);
  pthread_cond_destroy(&continue_cond);
  pthread_mutex_destroy(&continue_mutex);

  return(0);
}
```

Beispiel III

```
void prod (void* p) /* Producer thread */
{
  int i;
  for (i=0; i<100; i++) {
    printf("ping\n");

    pthread_mutex_lock(&arrived_mutex);
```

```
    arrived_flag = 1;
    pthread_cond_signal(&arrived_cond);
    pthread_mutex_unlock(&arrived_mutex);

    pthread_mutex_lock(&continue_mutex);
    while (continue_flag==0)
    pthread_cond_wait(&continue_cond,&continue_mutex);
    continue_flag = 0;
    pthread_mutex_unlock(&continue_mutex);
  }
}
```

Beispiel IV

```
void con (void* p) /* Consumer thread */
{
  int i;
  for (i=0; i<100; i++) {
    pthread_mutex_lock(&arrived_mutex);
    while (arrived_flag==0)
      pthread_cond_wait(&arrived_cond,&arrived_mutex);
    arrived_flag = 0;
    pthread_mutex_unlock(&arrived_mutex);

    printf("pong\n");

    pthread_mutex_lock(&continue_mutex);
    continue_flag = 1;
    pthread_cond_signal(&continue_cond);
    pthread_mutex_unlock(&continue_mutex);
  }
}
```

5.4 Threads und Objektorientierung

Threads und OO

- Offensichtlich sind Pthreads relativ unpraktisch zu programmieren
- Mutexes, Bedingungsvariablen, Flags und Semaphore sollten objektorientiert realisiert werden. Umständliche `init`/`destroy` Aufrufe können in Konstruktoren/Destruktoren versteckt werden
- Threads werden in Aktive Objekte umgesetzt
- Ein Aktives Objekt „läuft“ unabhängig von anderen Objekten

Thread Tools

```
#include<iostream>

#include"basicthread.hh"
#include"threadpool.hh"
#include"mutex.hh"
#include"barrier.hh"
#include"flag.hh"

TT::Mutex lock;
TT::Flag flag;
TT::Barrier barrier(10);

class Minimalistic : public TT::BasicThread
{
public:
  virtual void run () {
    flag.wait();
    lock.lock();
    std::cout << "thread␣running" << std::endl;
    lock.unlock();
  }
};

class SelfStartingWithArgs : public TT::BasicThread
{
public:
  SelfStartingWithArgs (int i) : rank(i) {
    this->start();
  }
  ~SelfStartingWithArgs () {
    this->stop();
  }
  virtual void run ()
  {
    lock.lock();
    std::cout << "thread␣" << rank << "␣running" << std::endl;
    lock.unlock();
    flag.signal();
  }
private:
  int rank;
};

class PoolElement : public TT::ThreadPoolElement<PoolElement>
{
public:
  virtual void run ()
  {
    barrier.sync();
    std::cout << "array␣thread␣" << this->context.i()
          << "␣of␣" << this->context.size() << std::endl;
```

```
    const PoolElement& next =
      this->context.thread(this->context.i()
          %this->context.size());
  }
};

int main (int argc, char *argv[])
{
  Minimalistic a; a.start();
  SelfStartingWithArgs b(5);
  TT::ThreadPool<PoolElement> c(10);
  c.start();
  a.stop();
  c.stop();
}
```

5.5 Ausklang

Thread Safety

- Darunter versteht man ob eine Funktion/Bibliothek von mehreren Threads gleichzeitig genutzt werden kann.
- Eine Funktion ist *reentrant* falls sie von mehreren Threads gleichzeitig gerufen werden kann.
- Eine Funktionen, die keine globalen Variablen benutzt ist reentrant
- Das Laufzeitsystem muss gemeinsam benutzte Resourcen (z.B. den Stack) unter wechselseitigem Ausschluss bearbeiten
- Der GNU C Compiler muss beim Übersetzen mit einem geeigneten Thread-Modell konfiguriert werden. Mit `gcc -v` erfährt man das Thread-Modell
- STL: Allokieren ist threadsicher, Zugriff mehrerer Threads auf einen Container muss vom Benutzer abgesichert werden.

Links

1 PThreads tutorial vom LLNL `https://computing.llnl.gov/tutorials/pthreads/`

2 Noch ein Tutorial `http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html`