Paralleles Rechnen

Olaf Ippisch
Institut für Mathematik
TU Clausthal
Erzstr. 1
D-38678 Clausthal-Zellerfeld
E-mail: olaf.ippisch@tu-clausthal.de

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

2. Februar 2017

Inhaltsverzeichnis

1 Einführung

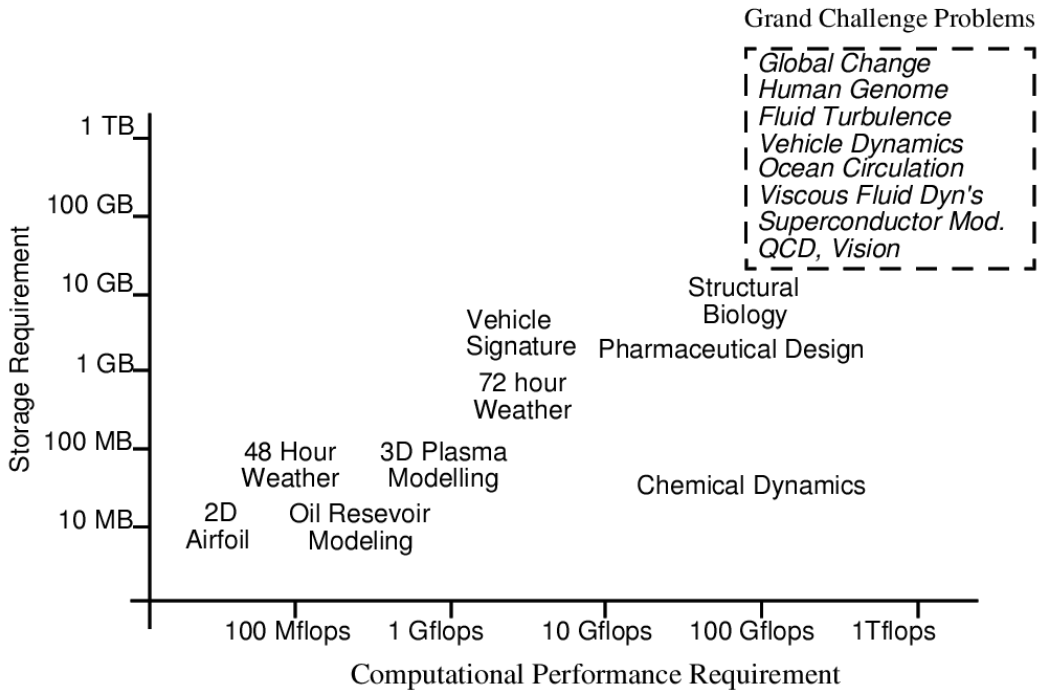
Inhalte der Vorlesung

- Architektur und Funktionsweise typischer Parallelrechner
- Shared-Memory Programmierung (OpenMP, Multi-threading)
- Message-Passing Programmierung (MPI)
- Bewertung paralleler Algorithmen
- Parallele Algorithmen (Lösung linearer Gleichungssysteme, FFT)
- Grafikkartenprogrammierung (CUDA, OpenCL)

Warum ist paralleles Rechnen wichtig?

- Seit einigen Jahre steigt die Geschwindigkeit einzelner Prozessorkerne kaum noch an
- Dagegen steigt die Anzahl an Prozessorkernen pro Chip stetig (selbst Smartphones haben heute bis zu acht Kernen)
- Der Bedarf an Rechenleistung ist – insbesondere für wissenschaftliches Rechnen – immer noch nicht befriedigt. Die Ansprüche sind dabei unterschiedlich:
  - Löse ein Problem gegebener (fester) Größe so schnell wie möglich Ziel: Minimiere Time-to-Solution
  - Löse ein möglichst großes Problem Ziel: Hohe Genauigkeit, komplexe Systeme
  - Löse sehr große Probleme so schnell wie möglich (oder in gegebener Zeit) sog. Grand Challenge Probleme

Grand Challenge Probleme



from Culler, Singh, Gupta: Parallel Computer Architecture

Klassifizierung Paralleler Probleme

Parallele Probleme lassen sich nach Ihrer Anforderung an Hauptspeicher und Rechenzeit in drei Gruppen einteilen

- Speicher begrenzt
- Rechenzeit begrenzt
- ausgewogen

Paralleles Rechnen ist Überall

- Multi-Tasking
  - Mehrere unabhängige Berechnungen ("threads of control") werden quasi-gleichzeitig auf einem einzigen Prozessor ausgeführt (time slicing)
  - Seit den 60er Jahren entwickelt um den Durchsatz zu erhöhen.
  - Interaktive Anwendungen erfordern es "viele Dinge gleichzeitig zu machen".
- Multi-Threading
  - Moderne Prozessoren haben viele Ausführungseinheiten ("execution units"), die meisten sogar mehrere Prozessorkerne.
  - Dadurch können mehrere Programme oder Programmteile (Threads) parallel ausgeführt werden.

  - "Hyperthreading" simuliert das Vorhandensein mehrere Kerne um eine optimale Auslastung der Ausführungseinheiten zu erreichen.

- Verteiltes Rechnen
  - Die Berechnung ist inhärent verteilt, weil die Informationen verteilt sind.
  - Beispiel: Eine internationale Firma oder Bank, Google.
  - Herausforderungen: Kommunikation zwischen verschiedenen Rechnerplattformen (Betriebsysteme, Rechnerarchitekturen), Portabilität und Sicherheit.

Verschiedene Stufen Parallelen Rechnens

- Paralleles Verhalten innerhalb eines Prozessorkerns ("instruction level parallelism")
- Hyperthreading
- Multi-core CPU's
- Grafikkarten
- Multi-Prozessor Shared-Memory Maschinen
- Parallele Computer Cluster, Supercomputer

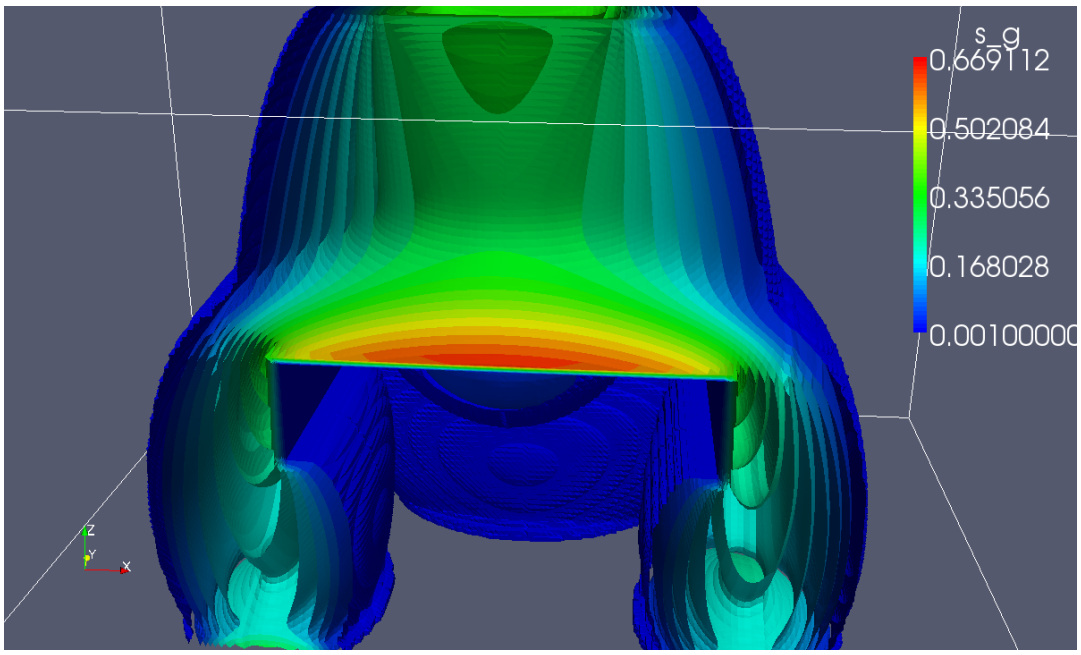
Effiziente Algorithmen sind von entscheidender Bedeutung!

- Rechenzeit für die Lösung eines (bestimmten) dünn besetzten linearen Gleichungssystems auf einem Computer mit 1 GFLOP/s.

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 d	10^{-2} s
$1 \cdot 10^6$	21 y	0.1 s
$1 \cdot 10^7$	21.000 y	1 s

- Parallelisierung rettet einen ineffizienten Algorithmus nicht.
- Für die Parallelisierung müssen Algorithmen mit einer guten sequentiellen Komplexität ausgewählt werden.

Problem mit fester Größe auf einem Multicore, Multi-Prozessorrechner
3D DNAPL Infiltration



Strong scaling, 3D, $160 \times 160 \times 96$

4×12 AMD Magny Cours, 2.1 GHz, 12×0.5MB L2, 12MB L3.

BiCGStab + AMG prec., ein einziger Zeitschritt

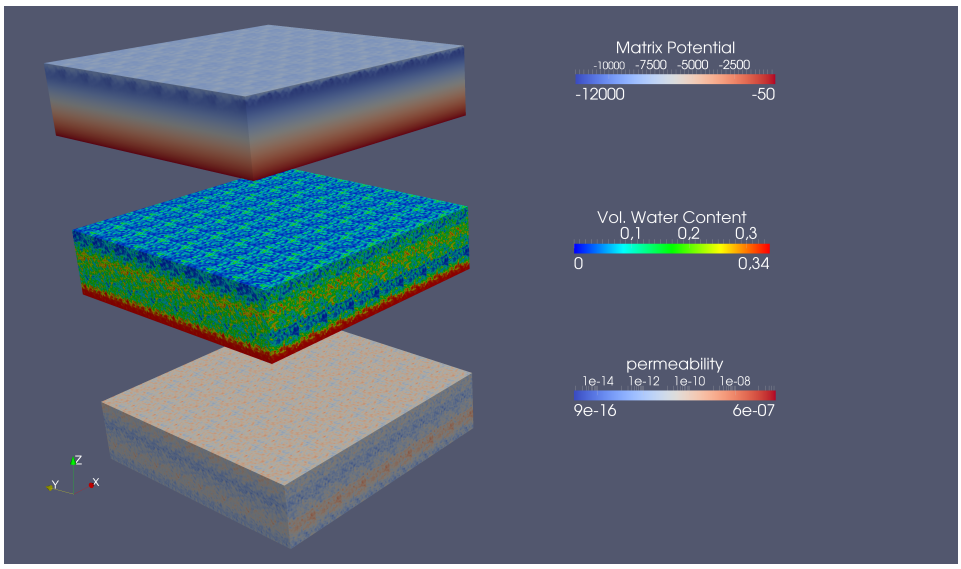
P	#IT(max)	T_{it}	S	T_{ass}	S	T_{total}	S
1	6.5	4.60	-	43.7	-	713.8	-
4	10	1.85	2.5	17.5	2.5	295.9	2.4
8	9	0.63	7.3	8.4	5.2	127.1	5.6
16	9.5	0.40	11.5	4.1	10.7	73.1	9.8
32	15	0.27	17.0	1.9	23.0	43.5	16.4

Vergleich mit Cray T3E von 1999

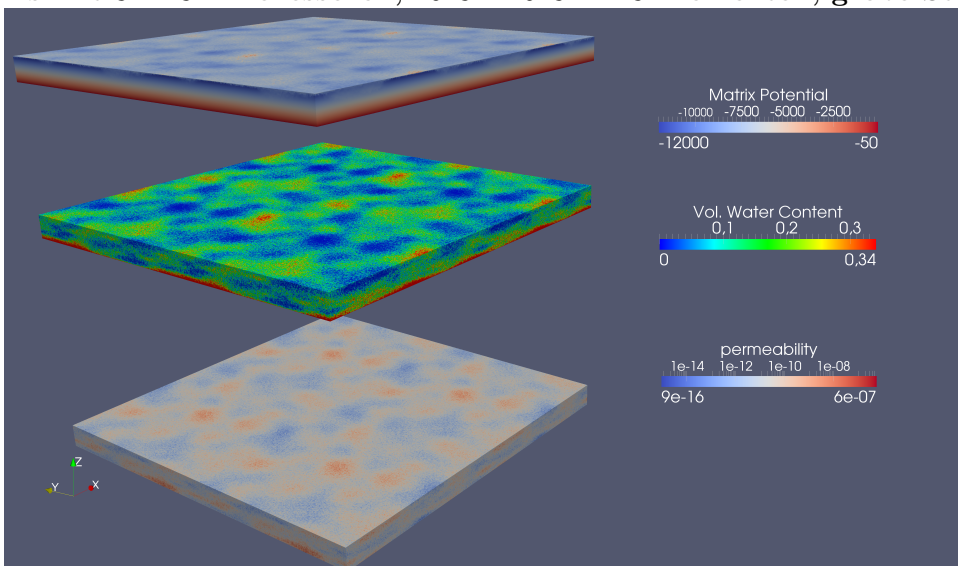
Maschine	# Unbekannte	Zeitschritte	Newtonschritte	T_{total}
256 Kerne T3E	2621440	50	264	14719
16 Kerne AMD	2457600	50	231	2500

Feste Anzahl Unbekannter pro Kern auf einem Supercomputer (Bluegene/Q)

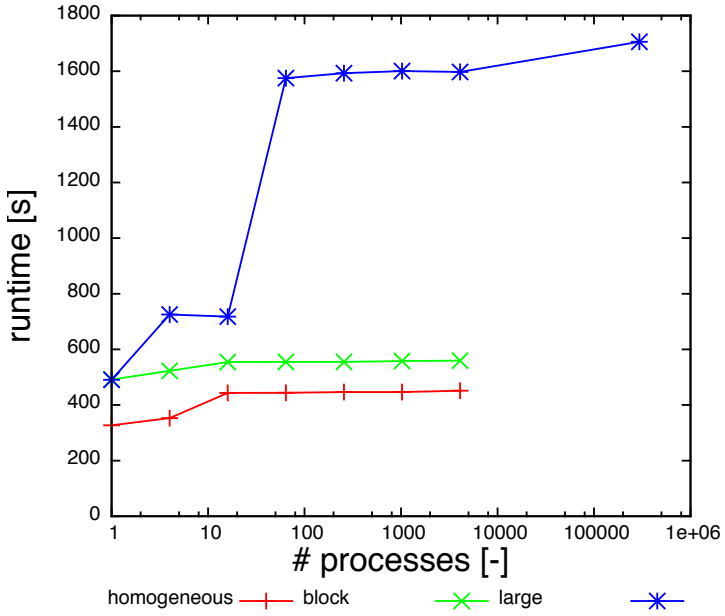
Ergebnis mit 8×8 *Prozessoren,* $512 \times 512 \times 128$ *Elementen, Blockstruktur*



Ergebnis mit 32×32 **Prozessoren,** $2048 \times 2048 \times 128$ **Elementen, große Struktur**

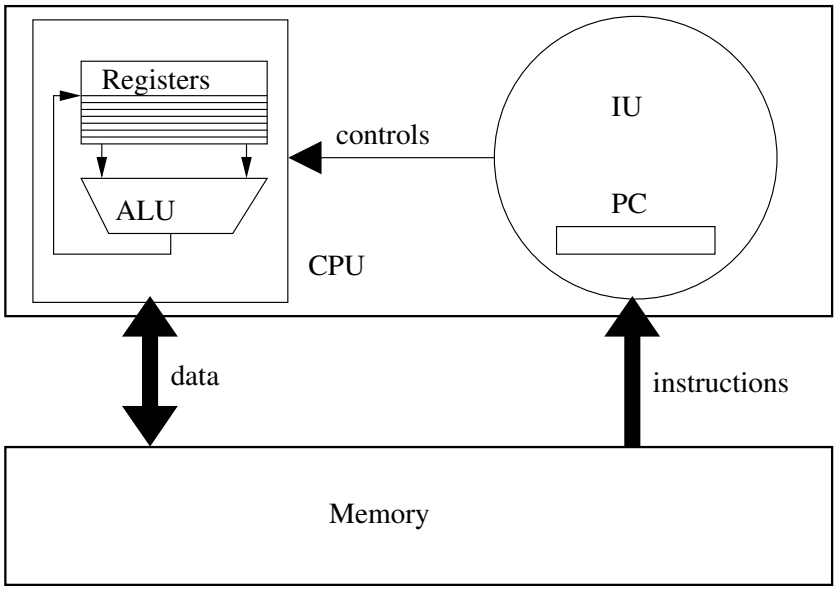


Weak scaling, 3D, $64 \times 64 \times 128$ Elemente/**Processor**


runtime [s]
1800
1600
1400
1200
1000
800
600
400
200
0
1
10
100
1000
10000
100000
1e+06
processes [-]
homogeneous
block
large

2 Prozessorarchitektur und Parallelismus auf Prozessorebene

Von Neumann Computer



IU: Instruction Unit

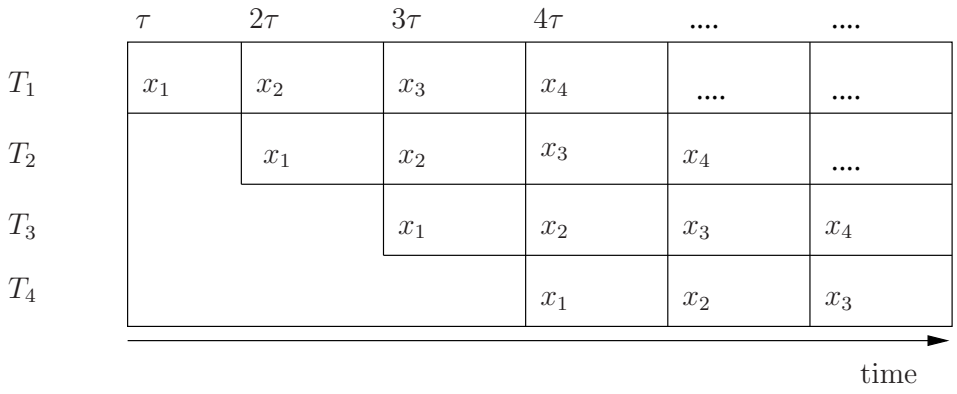
PC: Program Counter

ALU: Arithmetic Logic Unit

CPU: Central Processing Unit

2.1 Pipelining

Pipelining: Grundprinzip



- Task T kann in m Subtasks $T_1, \ldots, T_m$ unterteilt werden.
- Jeder Subtask benötigt die *gleiche* Zeit τ.

- Alle Subtasks sind *unabhängig* voneinander.
- Zeit zur Bearbeitung von N Tasks: Sequentielle Laufzeit: $T_S(N) = Nm\tau$ Parallele Laufzeit: $T_P(N) = (m + N - 1)\tau$.
- Speedup
  $S(N) = \frac{T_S(N)}{T_P(N)} = m\frac{N}{m+N-1}$.

Arithmetisches Pipelining

- Verwende Pipelining für Fließkomma-Operationen.
- Besonders geeignet für "Vektoroperationen' wie $s = x \cdot y$ oder $x = y + z$, da hier viele gleichartige, unabhängige Berechnungen auftreten.
- Daher der Name "Vektorprozessor".
- Erlaubt maximal $m = 10 \ldots 20$.
- Vektorprozessoren haben typischerweise eine sehr hohe Speicherbandbreite.
- Dies wird durch sog. *interleaved* Memory erreicht. Hier wird Pipelining of Speicherzugriffe angewendet.

Instruction Pipelining

- Verwende Pipelining bei der Verarbeitung von Maschinenbefehlen.
- Ziel: Verarbeite einen Befehl pro Zyklus.
- Typische Subtasks sind (m=5):
  - Holen eines Befehls (fetch).
  - Dekodieren des Befehls (decode).
  - Ausführen eines Befehls (execute).
  - Speicherzugriffe.
  - Zurückschreibe des Ergebnisses in ein Register.
- Reduced Instruction Set Computer (RISC): Verwende einfachen und einheitlichen Befehlssatz um Pipelining zu erleichtern (z.B.. load/store architecture).
- Bedingte Sprünge stellen ein Problem dar und erfordern einigen Aufwand z.B. für Sprungvorhersage (branch prediction units).
- Optimierende Compiler sind von entscheidender Bedeutung (instruction reordering, loop unrolling, etc.).

2.2 Superskalare Architekturen

- Die Anzahl an Transistoren auf einem Chip verdoppelt sich seit 1965 alle 18 bis 24 Monate (Moore'sches Gesetz)
- Ein typischer Desktop-Prozessor hat 2012 1–5 Milliarden Transistoren (Intel Core i7 Sandy Bridge Quad-Core Prozessor: 995 Millionen, Intel Xeon Haswell-EP 18-Core Prozessor: 5.69 Milliarden)
- Dies ging mit einer Verkleinerung der Transistorfläche (DRAM-Zelle 1990: 200 μm^2 Fäche, heute ca. 140 nm^2) und Strukturgröße (1990: 2 μm, heute: 22 nm) einher.
- Wurde dazu verwendet Prozessoren mit breiteren Registern (8-bit, 16-bit, 32-bit, 64-bit) und immer mehr Funktionseinheiten zu bauen (On-Chip FPU, mehrere ALUs, ...).
- Betrachte die Anweisungen

```
(1) a = b+c;
(2) d = e*f;
(3) g = a-d;
(4) h = i*j;
```

- Anweisungen 1, 2 and 4 können gleichzeitig ausgeführt werden, da sie unabhängig voneinander sind.
- Dies erfordert
  - Möglichkeit zur Ausführung mehrerer Anweisungen in einem einzigen Zyklus.
  - Mehrere Ausführungseinheiten.
  - Out-of-Order Execution.
  - Spekulative Ausführung.
- Ein Prozessor, der mehr als eine Anweisung pro Zyklus ausführen kann heißt "Superskalar"
- Wird ermöglicht durch:
  - Einen sogenannten "wide memory access" bei dem zwei Anweisungen gleichzeitig gelesen werden können.
  - "Very long instruction words".
- Mehrere Funktionsseinheiten mit Out-of-Order Execution wurden erstmals 1964 im CDC 6600 verwendet.
- Ein Parallelisierungsgrad von 3...5 kann erzielt werden.

2.3 SIMD und Hyper-threading

Klassifikation von Parallelrechnern nach Flynn (1972)

	Einzelner Datenstrom (Eine ALU)	Mehrere Datenströme (Mehrere ALUs)
Ein Anweisungsstrom, (Eine IU)	**SISD**	**SIMD**
Mehrere Anweisungsströme (Mehrere IUs)	—	**MIMD**

- **SISD** *single instruction single data*: Der von Neumann Rechner
- **SIMD** *single instruction multiple data*: Maschinen erlauben die gleichzeitige Ausführung einer Anweisung auf mehreren ALUs. Wichtige historische Maschinen: ILLIAC IV, CM-2, MasPar. Heute: SSE, AVX und CUDA!
- **MISD** *multiple instruction multiple data*: Diese Klasse ist leer
- **MIMD** *multiple instruction single data*: ist das dominante Konzept seit den frühen 90ern. (Fast) alle modernen Supercomputer sind von diesem Typ.

SIMD Erweiterungen auf Prozessorebene

Von MMX über SSE zu AVX

- MMX (MultiMedia eXtension) wurde 1996 von Intel eingeführt. Es erlaubte die gleiche Operation auf zwei 32-bit, vier 16-bit oder acht 8-bit integer Variablen anzuwenden.
- MMX wurde mit Hilfe der Register der Fließkommaeinheit realisiert.
- Der nächste Schritt war SSE (Streaming SIMD Extensions) im Jahr 1999. Sie führten spezielle 128-bit Register für SSE ein und erweiterten den Befehlssatz.
- SSE unterstützte ursprünglich nur singe-precision Fließkommaoperationen (im Gegensatz zu MMX). Integer und double precision Operationen wurden erst später (wieder)eingeführt (SSE2).
- Der Befehlssatz wurde durch SSE3, SSSE3 and SSE4 weiter vergrößert.
- Die neuesten Versionen sind XOP und CVT16 (zusätzliche Operationen), FMA4 (Operationen mit bis zu vier beteiligten Registern), alle von AMD, sowie AVX bzw. AVX-512 (zusätzliche Operationen, Operationen mit bis zu drei beteiligten Registern, 512-bit Register) von Intel.

Verwendung

- Durch das Setzen spezieller Optionen (beim GCC z.B. `-mmmx, -msse, -msse2, -msse3, -msse4.1, -msse4.2, -mavx, -mxop, -mfma4`) versucht der Compiler diese Erweiterungen automatisch zu nutzen.

- Die Optionen werden bei bestimmten `-march` Optionen automatisch aktiviert. `-march=corei7` wählt eine Intel Core i7 CPU mit Unterstützung für MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1 und SSE4.2.

- Eine optimale Performance wird erreicht, wenn der Programmcode speziell für die jeweilige Hardware geschrieben wurde (z.B. mit sogenannten SSE-Intrinsics).

Entwicklung der Taktfrequenz

- Die Taktfrequenz stieg bis 2003 stetig an, was zu einer deutlichen Beschleunigung der Prozessoren führte.

- Seitdem stagniert sie bei ca. 3 GHz, da bei höheren Taktfrequenzen der Leckstrom stark ansteigt. Eine Abschätzung der Verlustleistung P_{dyn} eines Prozessors ist:

  $$P_{\mathsf{dyn}} = \alpha \cdot C_{\mathsf{L}} \cdot V^2 \cdot f,$$

  α ist dabei eine Umschaltwahrscheinlichkeit, C_{L} die Belastungskapazität, V die Versorgungsspannung und f die Taktfrequenz. Da V linear mit f ansteigt, steigt P_{dyn} proportional zu f^3 (erster 32-bit Prozessor: 2 W, Intel Core i7 (Nehalem) mit 3.3 GHz: 103 W).

- Eine gewisse Entlastung ist durch eine dynamische Regelung der Taktfrequenz möglich (Turbo Boost).

Entwicklung der Hauptspeicherperformance

- Die Leistungsfähigkeit der Speicherchips (DRAM) stieg nicht im gleichen Maß. Die Latenz sank seit 1980 pro Jahr im Schnitt um 5 %, die Bandbreite stieg um etwa 10 %.

- Es ist zwar möglich schnelleren Speicher zu bauen, dieser ist jedoch für große Speichermodule zu teuer.

- Dadurch sind Prozessoren immer stärker durch die Speicherbandbreite begrenzt ("memory wall").

- Ein Hauptspeicherzugriff benötigt bei einem Intel Core i7 Prozessor heute etwa 180 Taktzyklen um ein 64-bit Wort aus dem DRAM Hauptspeicher zu holen, während er 1990 in 6 bis 8 Taktzyklen durchgeführt werden konnte.

- Für dieses Probleme gibt es zwei Lösungsansätze:
  - Die Simulation "virtueller Prozessoren" (Hyper-threading oder Simultaneous MultiThreading, SMT).
  - Die Einführung lokaler schneller Zwischenspeicher (Caches).

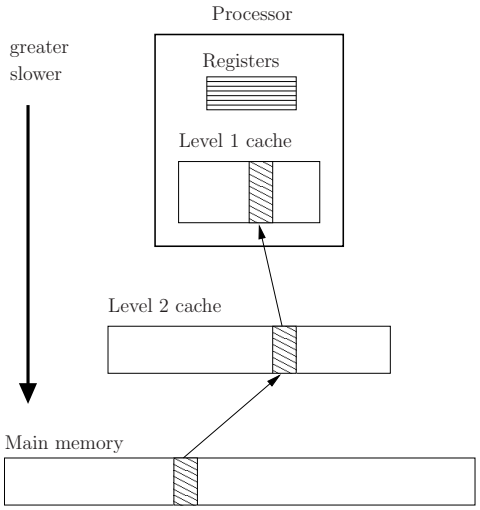
Hyper-threading / Simulateneous Multithreading

- Multithreading kann verwendet werden, um auch Funktionseinheiten auszulasten, die von einem Programm gerade nicht benötigt werden.
- Software Multithreading ist zu langsam und zu grobkörnig dafür. Eine Integration in die Prozessorhardware ist notwendig. Dies ist mit relativ kleinem Hardwareaufwand möglich (zusätzlicher Satz Register, Stackpointer und Programcounter).
- Wird heute von etlichen Prozessoren unterstützt. Viele Modelle der Intel Pentium und Xeon-Reihe unterstützen 2-fach Hyper-threading, die PowerPC A2-Prozessoren der Bluegene/Q-Reihe 4-faches.

2.4 Caches

Hierarchischer Cache:

- Überprüfe ob die Daten im Level l Cache sind. Wenn ja lade sie, sonst überprüfe den nächsthöheren Level.
- Wiederhole bis die Daten notfalls aus dem Hauptspeicher geholt werden.
- Daten werden in Form von *Cache Lines* von 32 . . . 128 Bytes (4 – 16 Worte) transportiert.



- Die effiziente Nutzung von Caches setzt *räumliche und zeitliche Lokalität* voraus.
- Vier Punkte müssen beim Design eines Caches beachtet werden:
  - *Platzierung*: Wo landet ein Block aus dem Hauptspeicher im Cache? Directly Mapped Cache, Assoziativer Cache.
  - *Identifizierung*: Wie findet man heraus, ob ein Datenblock schon im Cache ist?
  - *Replacement*: Welcher Datenblock wird als nächstes aus einem vollen Cache entfernt?
  - *Schreibstrategie*: Wie wird der Schreibzugriff gehandhabt? Write-through und write-back Caches.

- Es ist notwendig einen Programmcode an die Cache-Architektur anzupassen (cache-aware zu machen). Dies ist normalerweise nicht trivial und geht nicht automatisch.

- Caches können den Speicherzugriff verlangsamen, wenn auf die Daten in zufälliger Reihenfolge zugegriffen wird und sie nicht wiederverwendet werden.

Cache Organisation

- direkte Zuordnung: Hauptspeicherblock i kann nur in Platz $j = i \mod M$ im Cache geladen werden (M: Größe des Caches). Vorteil: einfache Identifikation, Nachteil: Cache scheinbar voll, obwohl es freien Platz gibt

- assoziativer Cache: Hauptspeicherblock i kann an jede Stelle im Cache geladen werden. Vorteil: flexible Cache-Nutzung, Nachteil: aufwändige Identifizierung (M Vergleiche, bzw. M Vergleichseinheiten)

- Kombination: $k - fach$ assoziativer Cache

Cache: Lokalitätsprinzip

- In normalen Programmen geht man davon aus, dass auf alle Speicherworte gleich schnell zugegriffen werden kann.

- Mit Cache gibt es aber Daten (die zuletzt geholten) auf die schneller zugegriffen werden kann.

- Dies muss im Algorithmus berücksichtigt werden.

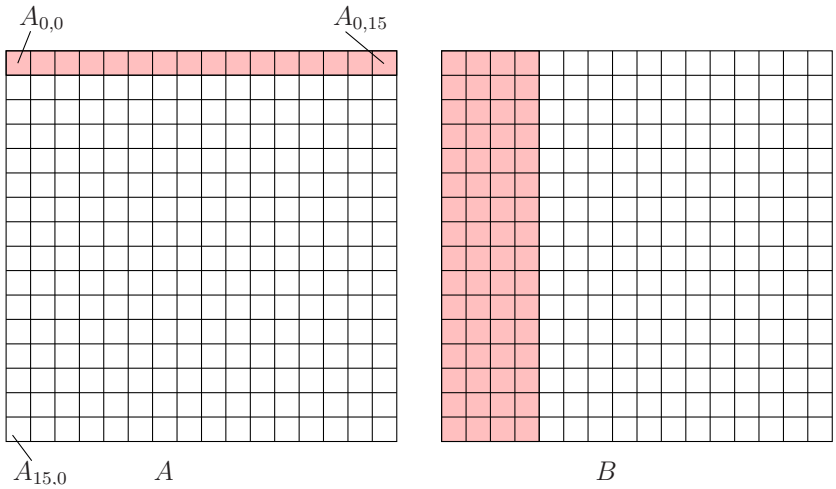
Cache Verwendung bei der Matrizenmultiplikation

- Multiplikation zweier $n \times n$-Matrizen $\mathbf{C} = \mathbf{AB}$

```
for (i=0;i<n;++i)
  for (j=0;j<n ;++j)
    for (k=0;k<n;++k)
      C[i][j] += A[i][k] * B[k][j];
```

- Angenommen eine Cache Line enthält vier Zahlen.

- A, B, C voll im Cache: $2n^3$ Rechenoperationen aber nur $3n^2$ Speicherzugriffe

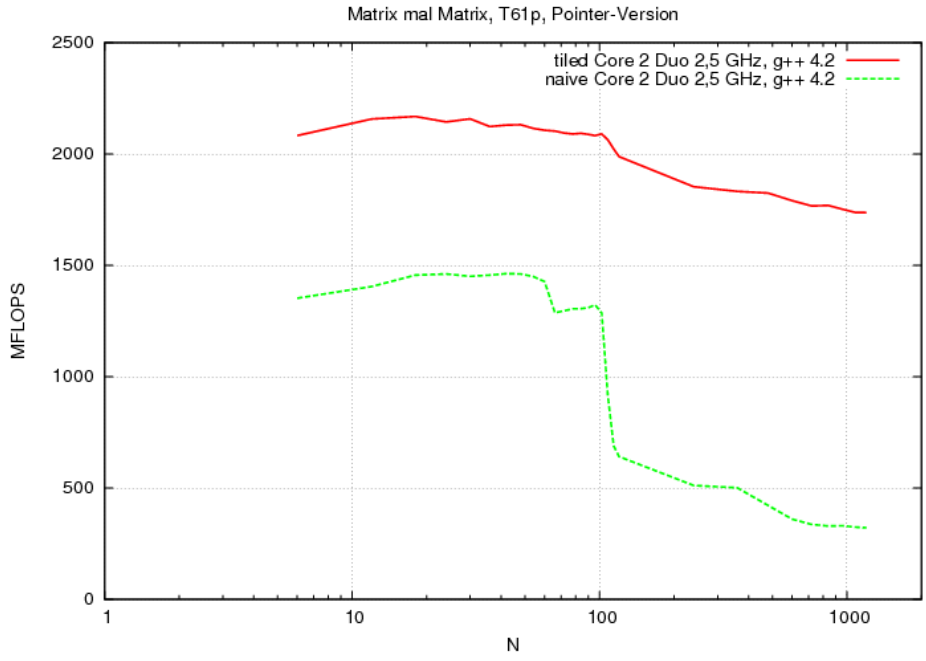


- Falls weniger als $5n$ Zahlen in den Cache passen: langsam
- Die Spalten von B werden nicht wiederverwendet.

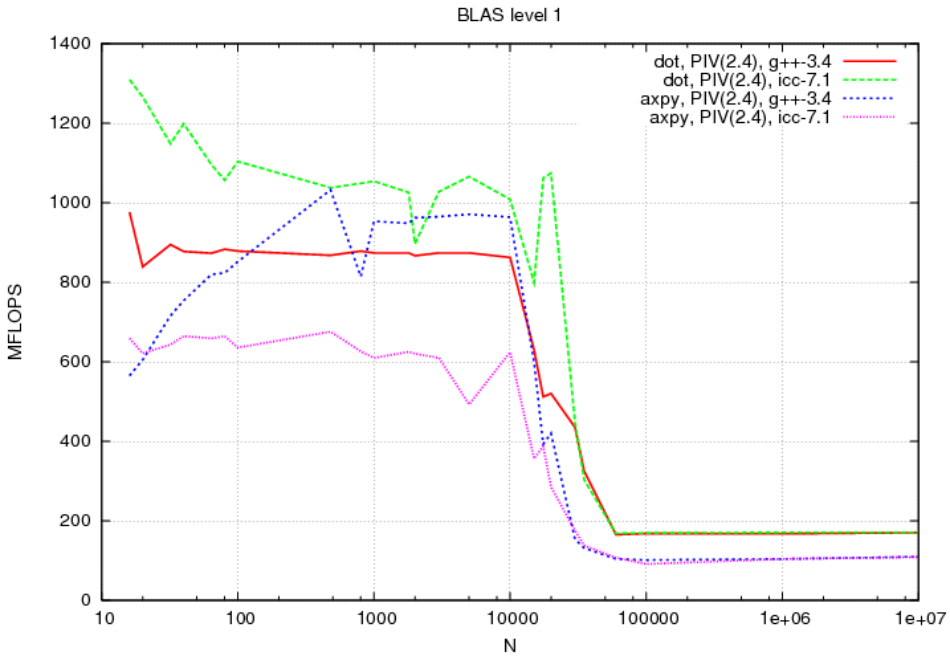
Tiling: Verarbeite Matrix in $m \times m$ Blöcken mit $3m^2 \leq M$

```
for (i=0;i<n;i+=m)
  for (j=0;j<n;j+=m)
    for (k=0;k<n;k+=m)
      for (s=0; s<m;++s)
        for (t=0; t<m;++t)
          for (u=0;u<m;++u)
            C[i+s][j+t] += A[i+s][k+u] * B[k+u][j+t];
```

Performance der Matrizenmultiplikation



BLAS1 Performance



Laplace-Problem

Laplace Gleichung:

$$\Delta u = 0$$

Finite-Differenzen Diskretisierung in 2D:

$$\begin{aligned}\Delta u(x_i, y_j) \approx & \frac{u(x_{i-1}, y_j) - 2u(x_i, y_j) + u(x_{i+1}, y_j)}{h^2} \\ & + \frac{u(x_i, y_{j-1}) - 2u(x_i, y_j) + u(x_i, y_{j+1})}{h^2}\end{aligned}$$

Gauß-Seidel-Iteration

Zur Lösung eines dünn-besetzten linearen Gleichungssystems

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

Algorithmus:

Sei $\mathbf{x}^{(k)}$ gegeben:

for $(i = 1; i \leq N; i = i + 1)$

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j<i} a_{ij} x_j^{(k+1)} - \sum_{j>i} a_{ij} x_j^{(k)} \right)$$

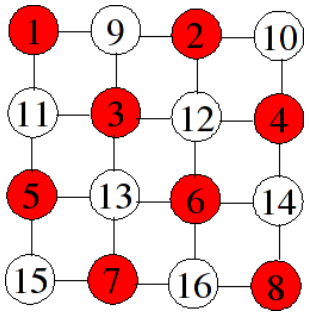
Ergebnis: $\mathbf{x}^{(k+1)}$

Verwende neue Lösung für Updates sobald verfügbar.

Erhöhung der Unabhängigkeit von Berechnungen

Durch eine andere Nummerierung der Unbekannten lässt sich der Gauß-Seidel-Algorithmus in 2 Teile mit völlig unabhängigen Berechnungen aufteilen:

Jeder Punkt im Gebiet erhält eine Farbe, so dass zwei Nachbarn nie die gleiche Farbe haben. Für strukturierte Gitter reichen zwei Farben aus (üblicherweise werden rot und schwarz verwendet). Die Unbekannten einer Farbe werden zuerst nummeriert, dann die nächste Farbe ....



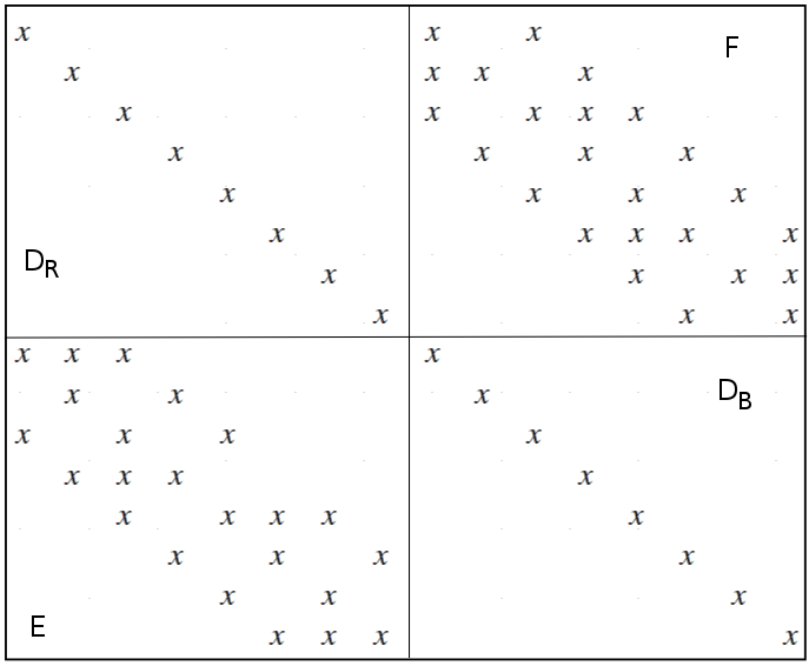
Red-Black-Ordering

Die Gleichungen für alle Unbekannten mit der gleichen Farbe sind dann unabhängig voneinander. Für strukturierte Gitter erhält man eine Matrix der Form:

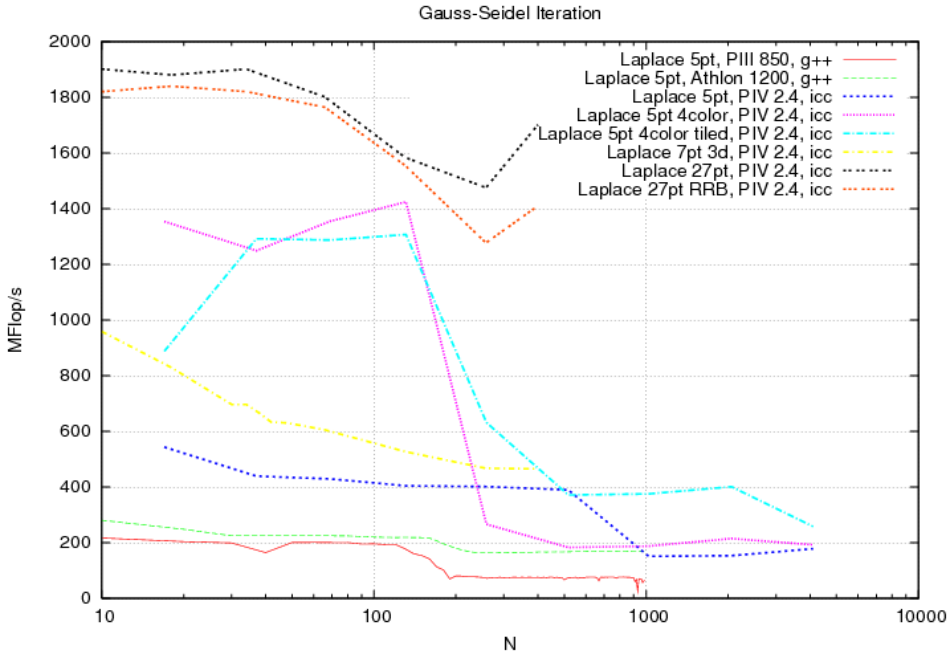
$$A = \begin{pmatrix} D_R & F \\ E & D_B \end{pmatrix}$$

Während solche Matrixtransformationen die Konvergenz von Lösern wie Steepest-Descent oder CG nicht beeinflussen (da diese nur von der Kondition der Matrix abhängt), kann sie die Konvergenz von Relaxationsverfahren wie Gauß-Seidel nachhaltig beeinträchtigen.

Beim RRB-Verfahren (Row-Wise Red Black) wird erst über die roten und dann über die schwarzen Unbekannten jeweils einer Zeile iteriert.



Laplace Performance



Laplace Performance

L1-Cachegrößen:

- Athlon: 64 KB Daten + 64 KB Instruktionen

- Pentium IV: 8 KB Daten + 12000 μOps (dekodierte Instruktionen)
- Skylake Core i7: 32 KB Daten und 32 KB Instruktionen pro Kern

L2-Cachegrößen:

- Athlon: 256 KB mit Prozessortakt
- Pentium IV: 512 KB mit Prozessortakt
- Skylake Core i7: 256 KB pro Kern

L3-Cachegrößen:

- Skylake Core i7: 8192 KB geteilt pro Prozessor

Taktfrequenzen:

- Athlon: 1200 MHz, MMX, 3DNow
- Pentium IV: 2400 MHz, MMX, SSE, SSE2
- Skylake Core i7: 3700(4000) MHz, SSE, SSE2, SSE3, SSSE3, SSE4, SSE4.1, SSE4.2 ADX, AVX, AVX2, AVX-512

2.5 Praxisblock: Vektorisierung

Auto-Vektorisierung

- Moderne Compiler können Code automatisch vektorisieren
- Beim GCC wird dies bei geeigneter Prozessorarchitekur ab `-O3` versucht, beim ICC ab `-O2`. Alternativ kann die Option `-ftree-vectorize` gesetzt werden.
- Keine Vektorisierung erfolgt beim GCC mit `-fno-tree-vectorize`, beim ICC mit der Option `-no-vec`
- Der Erfolg der Vektorisierung lässt sich beim GCC bis Version 4.8 mit `-ftree-vectorizer-verbose` ab Version 4.8 mit `-fopt-info-vec` ausgeben, beim ICC mit `-vec-report=<n>` (mit dem Parameter `<n>` lässt sich der Detailgrad der Ausgabe steuern). Beim GCC wird mit `-fopt-info-missed-vec` angezeigt, welche Schleifen warum nicht vektorisiert wurden.
- Geht auch für Fortran, wir behandeln hier aber nur C/C++.

Vektorisierbare Bestandteile

Vektorisierbar sind `for` Schleifen, die bestimmten Bedingungen genügen:

- Bekannte Anzahl Iterationen
  - Anzahl ist zu Beginn der Schleife bekannt (es reicht also, wenn diese zur Laufzeit festgelegt ist).
  - Kein bedingter Abbruch (`break` Anweisungen)
- Keine Verzweigungen

  - Keine `switch`-Anweisungen
  - Keine `if`-Anweisungen
  - Der `? :` Operator ist erlaubt.

- Nur die innerste Schleife kann vektorisiert werden. Der Compiler kann aber die Reihenfolge zur Optimierung umdrehen.
- Keine Funktionsaufrufe
  - Mathematische Funktionen wie `cos()`, `sin()`, `pow()`, `sqrt()` ...sind erlaubt.
  - Einige `inline` Funktionen sind auch erlaubt.

Nicht vektorisierbare Funktion

```
#include<vector>
#include<iostream>

int main()
{
  const std::size_t N=64;
  std::vector<long long> fib(N);
  fib[0]=0;
  fib[1]=1;
  for (std::size_t i=2;i<N;++i)
    fib[i] = fib[i-1] + fib[i-2];
  for (std::size_t i=0;i<N;++i)
    std::cout << fib[i] << std::endl;
}
```

Ergebnis ist von vorherigen Berechnungen abhängig.

```
#include<vector>
#include<iostream>
#include<cmath>

int main()
{
  const std::size_t N=65536;
  std::vector<double> root(N);
  for (std::size_t i=0;i<N;++i)
    root[i] = std::sqrt(i);
  for (std::size_t i=0;i<N;++i)
    std::cout << root[i] << std::endl;
}
```

Erste Schleife enthält Konversion von `int` nach `double`, diese ist nicht vektorisierbar. Zweite Schleife ist zu kompliziert (Methodenaufruf).

Vektorisierbare Funktion

```
#include<vector>
#include<iostream>
#include<cmath>

int main()
{
  const std::size_t N=65536;
  std::vector<double> root(N);
  for (std::size_t i=0;i<N;++i)
    root[i] = i;
  for (std::size_t i=0;i<N;++i)
    root[i] = std::sqrt(root[i]);
  for (std::size_t i=0;i<N;++i)
    std::cout << root[i] << std::endl;
}
```

Erste Schleife wird nicht vektorisiert. Zweite Schleife wird jetzt vektorisiert, da `sqrt()` ein erlaubter Funktionsaufruf ist.

Vektorisierung und Abhängigkeiten

- Read after Write

  ```
  for (size_t i=0;i<1024;i++)
    a[i] = a[i-1] + b[i];
  ```

  Nicht vektorisierbar. Variable wird erst geschrieben und dann gelesen. Es können daher nicht mehrere Werte auf einmal gelesen, manipuliert und dann zurückgeschrieben werden ohne das Ergebnis zu verfälschen.

- Write after Read

  ```
  for (size_t i=0;i<1024;i++)
    a[i] = a[i+1] + b[i];
  ```

  Vektorisierbar, da Ergebnis nur von dem Inhalt des Feldes zu Beginn der Berechnung abhängt. Variable wird erst gelesen und dann geschrieben. Mehrere Werte können auf einmal gelesen, manipuliert und dann zurückgeschrieben werden können ohne das Ergebnis zu verfälschen.

- Read after Read

  ```
  for (size_t i=0;i<1024;i++)
    a[i] = b[i%2] + c[i];
  ```

  Variable wird mehrfach gelesen. Keine echte Abhängigkeit, vektorisierbar.

- Write after Write

```
for (size_t i=0;i<1024;i++)
  a[i%2] = b[i] + c[i];
```

Variable wird mehrfach geschrieben. Da Ergebnis von der Reihenfolge der Schreibzugriffe abhängt nicht vektorisierbar.

Abhängigkeiten durch Aliasing

- Pointer können versteckte Abhängigkeiten enthalten:

```
void add(double *a, double *b, double *c, const size_t N)
{
    for (size_t i=0;i<1024;i++)
      a[i] = b[i] + c[i]
}
```

- Gibt es hier Abhängigkeiten?
- Wir könnten z.B. folgenden Funktionsaufruf machen:

```
add(a+1,a,c);
```

Dann haben wir eine Read after Write Abhängigkeit.

- Ist vom Compiler oft nicht einfach zu erkennen.

Restrict

- In C99 gibt es ein Keyword `restrict`, das dem Compiler mitteilt, dass sich ein Array mit keinem anderen überlappt:

```
void add(double * restrict a, double * restrict b, double *
   restrict c, const size_t N)
{
    for (size_t i=0;i<1024;i++)
      a[i] = b[i] + c[i]
}
```

- Die Verwendung von C99 muss z.B. mit `-std=c99` angeschalten werden.
- In C++ gibt es ein solches Keyword nicht.
- Es gibt aber Compiler eigene Direktiven, beim GCC z.B.

```
void add(double * __restrict__ a, double * __restrict__ b,
   double * __restrict__ c, const size_t N)
{
    for (size_t i=0;i<1024;i++)
      a[i] = b[i] + c[i]
}
```

- Wenn es doch eine Überlappung gibt, ist das Verhalten undefiniert. Deshalb ist Vorsicht geboten.

Speicherorganisation

- Die hohe Rechenkapazität durch Vektorisierung braucht auch einen sehr effizienten Speichertransfer
- Dazu sollten die benötigten Daten hintereinander im Speicher liegen.
- Bei mehrdimensionalen Feldern in C/C++ sollten daher zuerst über die letzte Koordinate iteriert werden.
- Array of Structs:

```
struct pointType
{
  double x, y, z;
};
pointType points[N];
```

  Häufig nicht effizient. Wird eine Koordinate nach der anderen abgearbeitet, dann liegen die benötigten Werte verteilt im Speicher. Die entsprechenden Anweisungen sind weniger effizient. Außerdem müssen eventuell nicht benötigte Daten in den Cache geladen werden ⇒ geringere effektive Speicherbandbreite, mehr Cache-Misses.

- Struct of Arrays

```
struct pointsType
{
  double x[N], y[N], z[N];
};
pointsType points;
```

  Jede Koordinate liegt hintereinander im Speicher, ist häufig deutlich effizienter.

Memory Alignment

- Aus dem Cache werden bei Vektorisierung immer eine ganze Reihe Werte geladen.
- Dies ist am effizientesten, wenn der erste Wert an einer Adresse liegt, die ein Vielfaches der Breite der SIMD/AVX-Register ist.
- Man sagt, die Variablen sind im Speicher “aligned”.
- Ist das nicht garantiert, verwendet der Compiler andere, langsamere Ladeinstruktionen.

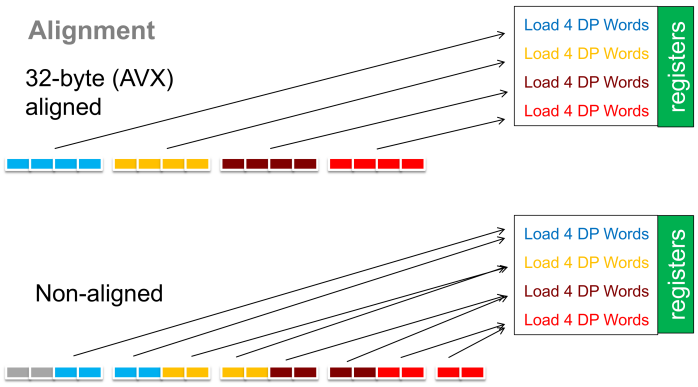


Bild: Aaron Birkland, Cornell University, Center for Advanced Computing

- Notwendiges Alignment
  - SSE2, SSE3, SSSE3, SSE4, SSE4a: 16 Byte (128 bit)
  - AVX, AVX2: 32 Byte (256 bit)
  - Intel ab Haswell, Xeon Phi: 64 Byte (512 bit)
- Der Compiler wählt häufig automatisch ein sinnvolles Alignment.
- In aufgerufenen Funktionen verlässt er sich aber oft nicht darauf und wählt eher defensiv die sichere Variante der Befehle ohne Alignment.
- Es gibt Compiler Hints, die dem Compiler sagen, dass ein Array ein bestimmtes Alignment hat. Beim GCC ist das `__builtin_assume_aligned(x,AL)` wobei das erste Argument die Variable ist und das zweite das Alignment

```
void add(double * __restrict__ a, double * __restrict__ b,
   double * __restrict__ c, const size_t N)
{
  __builtin_assume_aligned(a,32);
  __builtin_assume_aligned(b,32);
  __builtin_assume_aligned(c,32);
  for (size_t i=0;i<1024;i++)
    a[i] = b[i] + c[i]
}
```

- Gibt man ein Alignment an und ist dieses nicht vorhanden, kann dies zu einem "Segmentation Fault" führen.
- Alignment lässt sich auch erzwingen
  - In C++11 gibt es dafür die Anweisung `alignas(AL)`:

    ```
    alignas(64) int myArray[N];
    ```

  - Will man dynamische Speicherverwaltung betreiben gibt, dann gibt es dafür Funktionen wie `_mm_malloc` (nur Intel-Compiler), `_aligned_alloc` (C99) und `posix_memalign`

```
posix_memalign((void **)myArray,32,N*N*sizeof(float));
```

Freigabe erfolgt normal mit `free`

– Bei STL-Vektoren ist das etwas schwieriger und erfordert extra Allokatoren.

Alignment von mehrdimensionalen Feldern

- Bei mehrdimensionalen Arrays ist es evtl. sinnvoll das Array bis auf das gewünschte Alignment aufzufüllen ("Padding").
- Falls es für die zu vektorisierenden Operationen nicht auf die 2D-Reihenfolge ankommt, könnte man das Array auch wie folgt allozieren:

```
float **x = new float *[NX];
x[0] = new float [NX*NY];
for (size_t i=1;i<NX;++i)
  x[i] = x[i-1]+NX;
```

und dann direkt über `x[0]` iterieren.

- Oder man alloziert ein 1D-Array, z.B. einen Vektor und stellt den 2D-Bezug über die Indizes her:

```
std::vector x(NX*NY);
for (size_t i=1;i<NX;++i)
  for (size_t j=1;j<NY;++j)
    x[i*NY+j] = 1.;
```

Zum Vektorisieren kann man dann auch einfach über x iterieren.

C++-Vector mit aligned Allocator

```
#include <memory>
#include <vector>
#include <iostream>

#define ASSUME_ALIGNED(x,alignment) x =
   static_cast<decltype(x)>(__builtin_assume_aligned(x,alignment))

template<typename T, std::size_t alignment>
struct aligned_allocator
{

  typedef T value_type;

  template<typename T2>
  struct rebind
  {
    typedef aligned_allocator<T2,alignment> other;
```

```
  };

  T* allocate(std::size_t n)
  {
    std::cout << "Allocating with alignment " << alignment <<
       std::endl;
    T* r = nullptr;
    posix_memalign(reinterpret_cast<void**>(&r),alignment,n);
    return r;
  }

  void deallocate(T* t, std::size_t n)
  {
    free(t);
  }

};

int main()
{
  std::vector<char,aligned_allocator<char,128> > vec(16);
  auto start = vec.data();
  ASSUME_ALIGNED(start,128); // brauchst Du, damit der Compiler
     Bescheid weiss...
  std::cout << start << std::endl;
  return 0;
}
```

2.6 Performance von verschiedenen Matrixformaten

Naives Malloc

```
// allocate memory for matrices
void allocMatrix(double ***A, size_t n)
{
  // allocate row memory
  *A = (double **) malloc(n*sizeof(double *));

  // allocate column memory
  for (size_t i=0; i<n; i++)
    (*A)[i] = (double *) malloc(n*sizeof(double));

  // set to zero
  for(size_t i =0; i<n; i++)
    for(size_t j=0; j<n; j++)
      (*A)[i][j] = 0.0;
}
```

```
17 // release memory
   void freeMatrix(double **A, size_t n)
19 {
     // release rows first then columns
21   for (size_t i=0; i<n; i++)
       free(A[i]);
23   free(A);

25   // secure pointer
     A = 0;
27 }
```

Naives New

```
1  // allocate memory for matrices
   void allocMatrix2(double ***A, size_t n)
3  {
     // allocate row memory
5    (*A) = new double *[n];

7    // allocate column memory
     for (size_t i=0; i<n; i++)
9      (*A)[i] = new double[n];

11   // set to zero
     for(size_t i =0; i<n; i++)
13     for(size_t j=0; j<n; j++)
         (*A)[i][j] = 0.0;
15 }

17 // release memory
   void freeMatrix2(double **A, size_t n)
19 {
     // release rows first then columns
21   for (size_t i=0; i<n; i++)
       delete [] A[i];
23   delete [] A;

25   // secure pointer
     A = 0;
27 }
```

Verbessertes Malloc

```
   void allocMatrix3(double ***A, size_t n)
2  {
```

```
  // allocate row memory
  *A = (double **) malloc(n*sizeof(double *));

  // allocate column memory
  (*A)[0] = (double *) malloc(n*n*sizeof(double));
  for (size_t i=1; i<n; i++)
    (*A)[i] = (*A)[i-1]+n;

  // set to zero
  for(size_t i=0; i<n; i++)
    for(size_t j=0; j<n; j++)
      (*A)[i][j] = 0.0;
  //alternatively
  for(size_t i=0; i<n*n; i++)
    (*A)[0][i] = 0.0;
}

void freeMatrix3(double **A, size_t n)
{
  // release rows first then columns
  free(A[0]);
  free(A);

  // secure pointer
  A = 0;
}
```

Malloc 1D-Array

```
void allocMatrix4(double **A, size_t n)
{
  // allocate row memory
  *A = (double *) malloc(n*n*sizeof(double *));

  // set to zero
  for(size_t i=0; i<n; i++)
    for(size_t j=0; j<n; j++)
      (*A)[i*n+j] = 0.0;
  //alternatively
  for(size_t i=0; i<n*n; i++)
    (*A)[i] = 0.0;
  }

// release memory
void freeMatrix4(double *A, size_t n)
{
  // release rows first then columns
```

```
19   free(A);

21   // secure pointer
     A = 0;
23 }
```

Naive Verwendung von C-Arrays

```
   void useMatrix(size_t n)
2  {
     double A[n][n];
4    // set to zero
     for(size_t i=0; i<n; i++)
6      for(size_t j=0; j<n; j++)
         A[i][j] = 0.0;
8  }

10
   void useMatrix2()
12 {
     double A[500][500];
14   // set to zero
     for(size_t i=0; i<500; i++)
16     for(size_t j=0; j<500; j++)
         A[i][j] = 0.0;
18 }
```

Vektoren

```
   void useMatrix3(size_t n)
2  {
     std::vector<std::vector<double> > A(n);
4    for(size_t i=0; i<n; i++)
       A[i].resize(n);
6    // set to zero
     for(size_t i=0; i<n; i++)
8      for(size_t j=0; j<n; j++)
         A[i][j] = 0.0;
10 }

12
   void useMatrix4(size_t n)
14 {
     std::vector<double> A(n*n);
16   // set to zero
     for(size_t i=0; i<n; i++)
18     for(size_t j=0; j<n; j++)
```

```
        A[i*n+j] = 0.0;
20  // alternatively
    std::vector<double> B(n*n,0.0);
22 }
```

Performance Messprogramm

```
  inline double mflop(long n)
2 {
    return 2.0 * ((double) (n * n * n)) * 1E-6;
4 }

6 void mmProdVanilla(size_t n, size_t repeat)
  {
8   // pointers to matrix structures
    double** A = NULL;
10  double** B = NULL;
    double** C = NULL;
12  // allocate memory
    allocMatrix3(&A, n);
14  allocMatrix3(&B, n);
    allocMatrix3(&C, n);
16
    // init matrices
18  for(size_t i=0; i<n; i++)
      for(size_t j=0; j<n; j++)
20      A[i][j] = B[i][j] = (double) (i+j);
    // run simulation with time measurement
22  auto tstart = std::chrono::steady_clock::now();
    for(size_t r=0; r<repeat; r++)
24    for(size_t i=0; i<n; i++)
        for(size_t j=0; j<n; j++)
26        for(size_t k=0; k<n; k++)
            C[i][j] += A[i][k] * B[k][j];

    auto tend = std::chrono::steady_clock::now();
2   auto tt = std::chrono::duration<double> (tend-tstart).count();

4   std::cout << C[0][0] << std::endl;;
    // analyze results (n, MFLOPs, t_proc, repeats)
6   std::cout << n << "␣" << repeat*mflop(n)/tt << "␣" << tt << "␣
       " << repeat << std::endl;
    fflush(stdout);
8
    // free memory
10  freeMatrix3(A, n);
    freeMatrix3(B, n);
```

```
12   freeMatrix3(C, n);

14   return;
}
```

Performance (MFLOP/s)

```
Matrix-Dim    500   MFLOP/s -O3     -O0
Malloc                      1233    266
Verbessertes Malloc         1779    332
Malloc 1D-Array             2123    372
A[n][n]                     1855    405
A[500][500]                 2133    494
vector                       940    115
vector 1D-Array             1730    216

Matrix-Dim    500   MFLOP/s -O3     -O0
Malloc                      1233    266
Verbessertes Malloc         1779    332
Malloc 1D-Array             2123    372
vector 1D-Array             1730    216

Matrix-Dim    1000          -O3
Malloc                      281
Verbessertes Malloc         1116
Malloc 1D-Array             1245
vector 1D-Array             1095

Matrix-Dim    1500          -O3
Malloc                      252
Verbessertes Malloc         248
Malloc 1D-Array             263
Vector 1d-Array             260
```

2.7 Multi-Core Prozessoren

- Multi-Core Prozessoren lösen/umgehen mehrere Probleme
  - Die Geschwindigkeit eines einzelnen Prozessors stagniert, da sich die Prozessorfrequenz aufgrund des steigenden Stromverbrauchs nicht weiter steigern lässt (Power Wall)
  - Neben dem Stromverbrauch ist auch die Signallaufzeit begrenzend. Bei einem 3 GHz kann ein Signal in einem Taktzyklus maximal 10 cm transportiert werden. Durch die Größe eines Prozessors wird wiederum die maximale Anzahl von Kontakten zum Anschluss des Speichers begrenzt, was die Speicherbandbreite limitiert. Die Lücke zwischen der Prozessorgeschwindigkeit und der Zeit für Hauptspeicherzugriffe wächst (Memory Wall)
  - Die Integration von immer mehr Funktionseinheiten in einen Prozessorkern stößt an Grenzen, da inhärente Abhängigkeiten in Programmen ihre effiziente Ausnutzung verhindern. Auch die Pipelines lassen sich nicht beliebig verlängern (ILP Wall).
- Im Vergleich zu einem Multi-Prozessor System bieten Multi-Core Prozessoren
  - eine schnellere Kommunikation zwischen den Kernen
  - einen niedrigeren Stromverbrauch
  - ein einfacheres Platinenlayout
- Der erste Dual-Core Prozessor Power-4 wurde von IBM im Jahr 2001 vorgestellt.
- Durch Intel und AMD erreichten Multi-Core Prozessoren 2005 den Massenmarkt .
- Bis zu 18 Kerne auf einem Prozessor sind heute verfügbar, Prozessoren mit bis zu 1000 Kernen werden in den nächsten Jahren erwartet. Intel Phi Prozessoren verfügen über bis zu 72 Kerne.

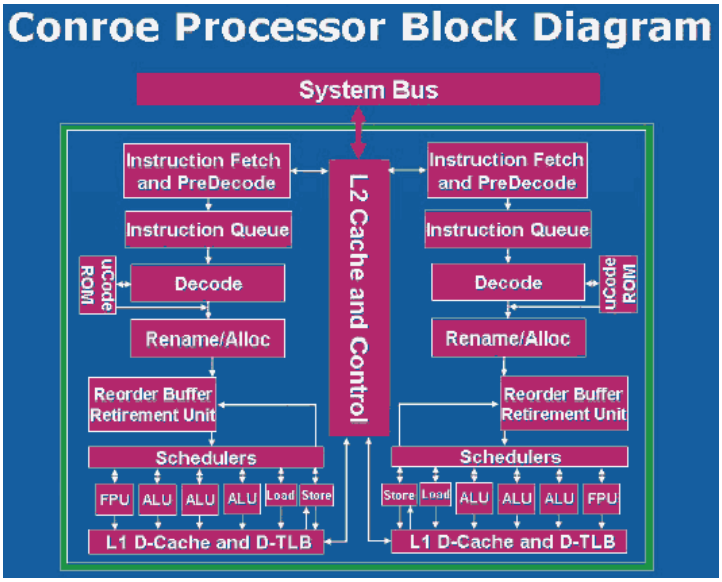
Hauptunterschiede zwischen verschiedenen Multi-Core Prozessorarchitekturen

- Größe und Organisation des Caches
  - Mit oder ohne L3 Cache
  - Geteilt oder separat
  - ...
- Speicherzugriff
  - Integrierter oder separater Memory Controller
  - Anzahl integrierter Memory Controller
  - ...
- Verbindung der einzelnen Multi-Core Prozessoren in einem Multi-Prozessor System
  - Systembus
  - Direkte Verbindung (AMD Hypertransport, Intel Quickpath Interconnects)
  - ...

2.8 Beispiele

Intel Conroe Architektur

2007 eingeführt



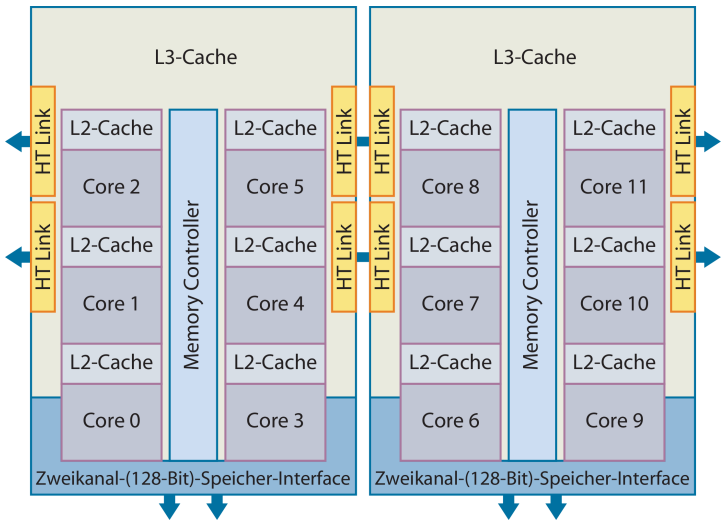
von Intel

Intel Allendale Dual-Core Prozessor

- L1 Cache: 32 KB für Instruktionen, 32 KB für Daten.
- L2 Cache: 1 MB geteilt
- Superscalarity
  - Instruction Decoder kann vier Anweisungen pro Taktzyklus dekodieren.
  - Integer Units 3-fach, FP Unit (3 OPS pro Taktzyklus).
  - Out-of-Order Execution, Branch Prediction.
- Pipelining: 14 Stages.
- MMX, SSE, SSE2, SSE3, SSSE3

AMD Magny-Cours Architektur

2010 eingeführt



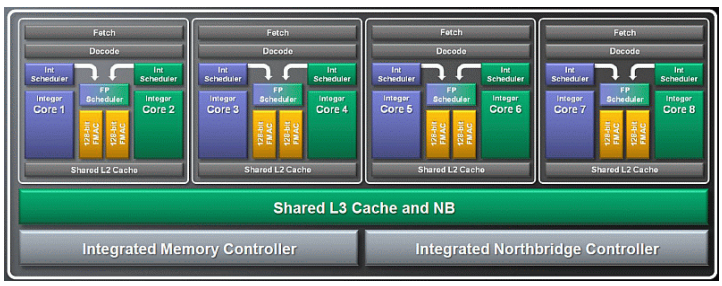
aus c't, Ausgabe 9, 2010

AMD Magny-Cours 12-Kern Prozessor

- Multi-chip Modul, zwei 6-Core Chips
- L1 Cache: 64 KB für Instruktionen, 64 KB für Daten.
- L2 Cache: 12×512 KB
- L3 Cache: 2×6 MB, geteilt
- Superscalarity
  - Instruction Decoder, Integer Units, FP Units, Address Generierung 3-fach (3 OPS pro Taktzyklus).
  - Out-of-Order Execution, Branch Prediction.
- Pipelining: 12 Stages Integer, 17 Stages Fließkommazahlen
- MMX, Extended 3DNow!, SSE, SSE2, SSE3, SSE4a
- Ein integrierter Memory-Controller pro Chip, 128-bit Speicherkanäle.
- HyperTransport: kohärenter Zugriff auf nicht-lokalen Speicher.

AMD Bulldozer Architektur

2011 eingeführt



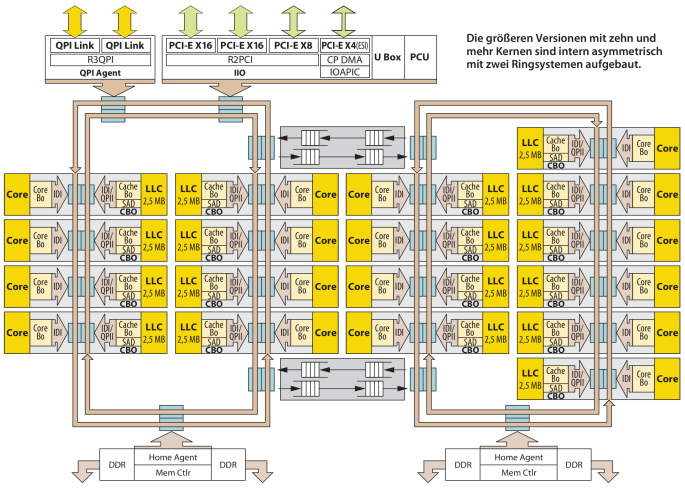
von AMD

AMD Abu Dhabi 16-Kern Prozessor

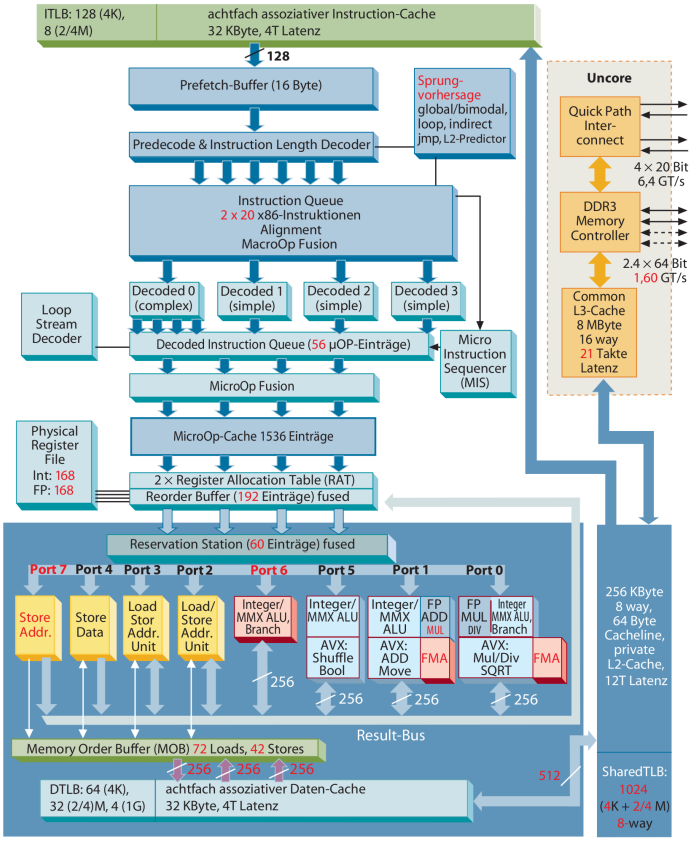
- Multi-chip Modul, zwei 8-Core Chips
- L1 Cache: 64 KB für Instruktionen (geteilt von 2 Kernen), je Kern 16 KB für Daten.
- L2 Cache: 8×2048 KB (geteilt von 2 Kernen)
- L3 Cache: 2×8 MB, geteilt, langsamer getaktet
- Superscalarity
  - Instruction Decoder, Integer Units (2-fach), FP Units (1 pro zwei Kerne), Address Generierung.
  - Out-of-Order Execution, Branch Prediction.
- Pipelining: 12 Stages Integer, 17 Stages Fließkommazahlen
- MMX, Extended 3DNow!, SSE, SSE2, SSE3, SSE4a, SSE4.2, AES, CLMUL, AVX, XOP, FMA4, FMA3
- Ein integrierter Memory-Controller pro Chip, 128-bit Speicherkanäle.
- HyperTransport
- Teilweise integrierter Grafikchip

Intel Haswell Architektur

2014 eingeführt



aus c't, Ausgabe 21, 2014



aus c't, Ausgabe 14, 2013

Intel Haswell 18-Kern Prozessor

- 8 und 10-Kern Ringe verbunden über zwei High-Speed Switches auf einem Chip
- L1 Cache: 32 KB für Instruktionen, 32 KB für Daten.
- L2 Cache: 256 KB
- L3 Cache: 8 MB, geteilt
- teilweise mit 128 MB L4 Cache
- Superscalarity
  - Bis zu 8 OPS pro Taktzyklus
  - Out-of-Order Execution, Branch Prediction.
- Pipelining: 14-19 Stages
- MMX, SSE, SSE2, SSE3, SSE4.2, AES, AVX, AVX2, FMA3
- Zwei Memory-Controller pro Chip
- QPI Link
- Integrierter Grafikchip

Vergleich AMD/Intel

Caches und TLBs

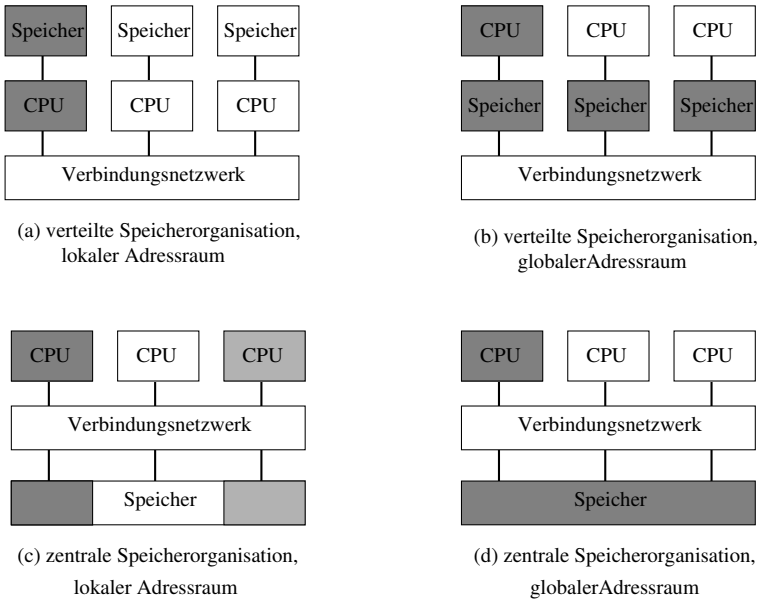
	Nehalem (2700K)	Ivy Bridge (3770 K)	Haswell (4770K)	AMD Piledriver 2x (FX 8350)
L1I: L1-Instruktions-Cache	32KB, 4-way	32KB, 8-way	32KB, 8-way	64KB, 2-way für 2 Kerne
L1D: L1-Daten-Cache	32KB 8-way	32KB 8-way	32KB 8-way	16 KB 4-way WT
L1D-Latenz (Load-to-Use)	4 Takte	4 Takte	4 Takte	4 Takte
L1D-Lesebandbreite	16 Bytes/Takt	32 Bytes/Takt	64 Bytes/Takt	32 Bytes/Takt
L1D-Schreibbandbreite	16 Bytes/Takt	32 Bytes/Takt	32 Bytes/Takt	16 Bytes/Takt
L2: Gemeinsamer Cache	256KB, 8-way, Inklusiv	256KB, 8-way, Inklusiv	256KB, 8-way, Inklusiv	2M 16-way für 2 Kerne Inkl./Exkl.
L2-Latenz (Load-to-Use)	10 Takte	11 Takte	11 Takte	20 Takte
L2-Bandbreite zu L1	32 Bytes/Takt	32 Bytes/Takt	64 Bytes/Takt	32 Bytes/Takt
L3-Gemeinsamer Cache aller Kerne	8 MByte 16-way	8 MByte 16-way	8 MByte 16-way	8 MByte 64-way
L3-Latenz (Load-to-Use)	35-40 Takte	25-31 Takte	21 Takte	55-60 Takte
L1-ITLB:	4K: 128, 4-way, 2M/4M: 7/Thread	4K: 128, 4-way, 2M/4M: 8/Thread	4K: 128, 4-way, 2M/4M: 8/Thread	4K: 48 full, 2/4M: 24 full
L1-DTLB:	4K: 64, 4-way, 2M/4M: 32 4-way 1G: fractured	4K: 64, 4-way, 2M/4M: 32 4-way 1G: 4, 4-way	4K: 64, 4-way, 2M/4M: 32 4-way 1G: 4, 4-way	4K: 64 full, 2/4M: 64 full
L2-TLB: Gemeinsamer TLB	4K: 512, 4-way	4K: 512, 4-way	4K+2M: shared: 1024, 8-way	L2D: mixed : 1024 8-way, L2I 4K: 512 4-way, L2I 2/4M: 1024 8-way
L1/L2-Cacheline	64 Byte	64 Byte	64 Byte	64 Byte

aus c't, Ausgabe 14, 2013

3 Multiprozessorsysteme

3.1 Speicherorganisation

Skalierbare Rechnerarchitekturen



(a) verteilte Speicherorganisation, lokaler Adressraum

(b) verteilte Speicherorganisation, globalerAdressraum

(c) zentrale Speicherorganisation, lokaler Adressraum

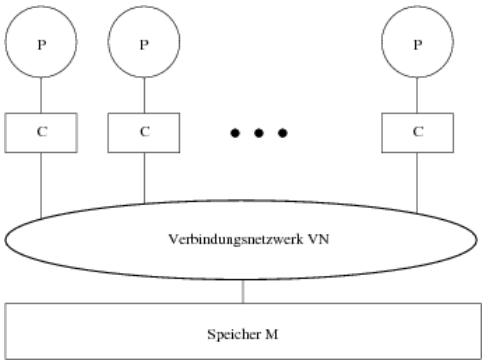
(d) zentrale Speicherorganisation, globalerAdressraum

Unterscheidung nach Anordnung von Prozessor, Speicher und Kommunikationsnetzwerk

Klassifikation nach der Art des Datenaustausches:

- Gemeinsamer Speicher (Shared Memory)
  - *UMA – uniform memory access.* Gemeinsamer Speicher mit uniformer Zugriffszeit.
  - *NUMA – nonuniform memory access.* Gemeinsamer Speicher mit *nicht*-uniformer Zugriffszeit, mit Cache-Kohärenz spricht man auch von *ccNUMA*.
- Verteilter Speicher (Distributed Memory)
  - *MP – multiprozessor.* Privater Speicher mit Nachrichtenaustausch.

Shared Memory: UMA

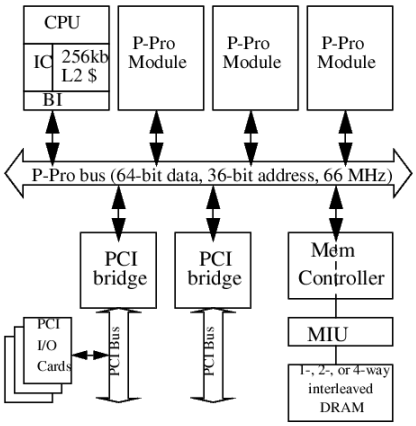


- *Globaler Adressraum*: Jedes Speicherwort hat global eindeutige Nummer und kann von allen Prozessoren gelesen und geschrieben werden.
- Zugriff auf Speicher erfolgt über *dynamisches* Verbindungsnetzwerk welches Prozessor und Speicher verbindet (davon später mehr).
- Speicherorganisation: *Low-order interleaving* – aufeinanderfolgende Adressen sind in aufeinanderfolgenden Modulen. *High-order interleaving* – aufeinanderfolgende Adressen sind im selben Modul.

Cache ist notwendig um

- Prozessor nicht zu bremsen, und
- Last vom Verbindungsnetzwerk zu nehmen.
- Cache-Kohärenzproblem: Ein Speicherblock kann in mehreren Caches sein. Was passiert, wenn ein Prozessor schreibt?
- Schreibzugriffe auf den selben Block in verschiedenen Caches müssen sequentialisiert werden. Lesezugriffe müssen stets aktuelle Daten liefern.
- UMA erlaubt den Einsatz von bis zu wenigen 10er Prozessoren.

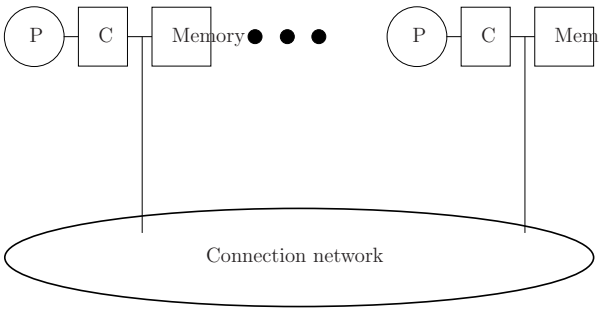
Shared Memory Board: UMA



Quad-Prozessor Pentium Pro Motherboard

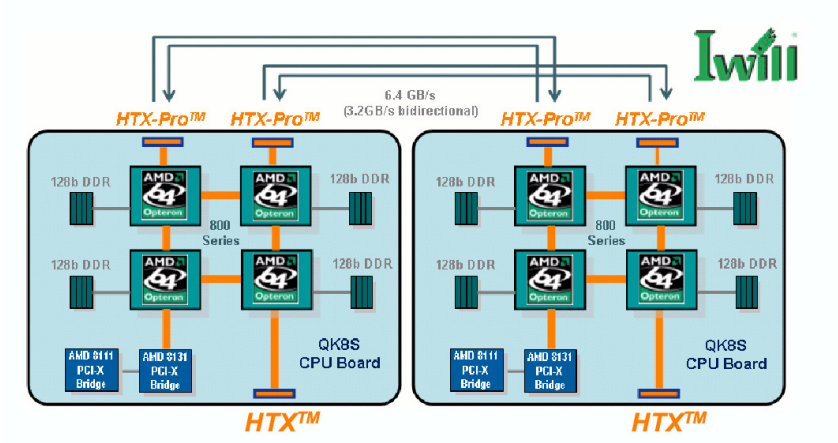
- Symmetric multi processing (SMP)
- Zugriff zu jedem Speicherwort in gleicher Zeit
- Unterstützung von Cache Kohärenz Protokollen (MESI)

Shared Memory: NUMA



- Jeder Baustein besteht aus Prozessor, Speicher und Cache.
- *Globaler Adressraum*: Jedes Speicherwort hat global eindeutige Nummer und kann von allen Prozessoren gelesen und geschrieben werden.
- Zugriff auf lokalen Speicher ist schnell, Zugriff auf andere Speicher ist (wesentlich) langsamer, aber transparent möglich.
- Cache-Kohärenzproblem wie bei UMA
- Extreme Speicherhierarchie: level-1-cache, level-2-cache, local memory, remote memory
- Maschinen bis etwa 4000 Prozessoren (SGI UV 1000)

Shared Memory Board: NUMA

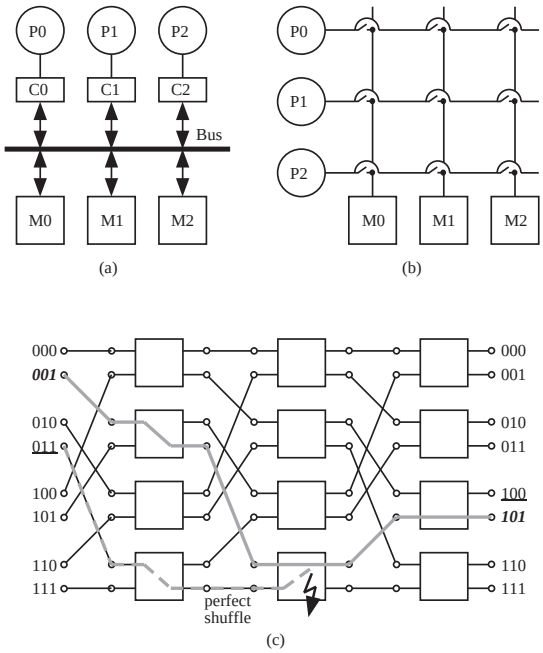


Quad-prozessor Opteron Motherboard

- Verbindung mit Hypertransport HTX-Technologie
- Nicht-uniformer Speicherzugriff (NUMA)

Dynamische Verbindungsnetzwerke

Leitungsvermittelnd: Elektrische Verbindung von Quelle zum Ziel.



(a) Bus: verbindet nur jeweils zwei Einheiten zu einer Zeit, also nicht skalierbar. Vorteile: billig, Cache-Kohärenz durch Schnüffeln.
(b) Crossbar: Volle Permutation realisierbar, aber: P^2 Schaltelemente.
(c) Ω-Netzwerk: $(P/2)\,\mathrm{ld}\,P$ Schaltelemente, keine volle Permutation möglich, jede Stufe ist *perfect shuffle*, einfaches Routing.

Hypertransport und QPI

- Verbindung der Prozessoren auf einem Mainboard erfolgt über High-Speed-Verbindungen: Intel QuickPath Interconnect (QPI) oder AMD Hypertransport (HT)
- Beides sind serielle Verbindungen um den Effekt unterschiedlich langer Leitungsbahnen zu minimieren
- Bis zu 4.8 GHz Taktfrequenz bei QPI und 3.2 GHz bei HT
- Übertragung bei steigender und fallender Taktflanke ⇒ bis zu 9.6 bzw. 6.4 GT/s (Gigatransfer per second)
- Bündel aus 20 (QPI) bis 32 (HT) Leitungen pro Verbindung
- Übertragungsraten bis zu 35.6 GB/s (QPI) bzw. 51.2 GB/s (HT)
- Mehrere Protokollschichten
  - Physical Layer: Reiner Datentransport
  - Link Layer (QPI), Switching Layer (HT): Zusammensetzen von Paketen
  - Routing Layer: Weiterleitung von Daten in komplexeren Rechnern (mehrere CPUs)
  - Protocol Layer: Versendung gesamter Cache Lines, Herstellung von Cache-Kohärenz (Home oder Source Snooping)

NUMA-Latenz bei Intel E5-4600

- Messung mit 4 Prozessorserver, Intel E5-4600 Prozessoren, 2 QPI Links pro Prozessor [1]
- Zugriffszeiten in ns auf Speicher angebunden an CPU n

CPU	0	1	2	3
0	72	291	323	294
1	296	72	293	315
2	319	296	71	296
3	290	325	300	71

- Lokaler Speicher sehr schnell
- Zugriffszeit abhängig von der Entfernung im Netzwerk (direkt angebunden, 1 QPI Link oder 2 QPI Links entfernt)
- Maximaler Unterschied etwa Faktor 4.5

NUMA-Latenz bei Intel E7-4800

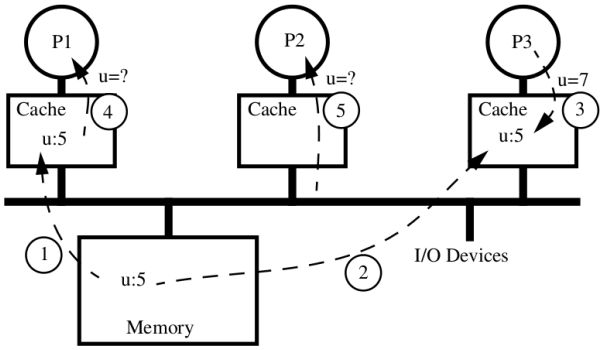
- Messung mit 4 Prozessorserver, Intel E7-4800 Prozessoren, 4 QPI Links pro Prozessor, davon 3 für Verbindungen mit anderen Prozessoren [2]
- Zugriffszeiten in ns auf Speicher angebunden an CPU n

CPU	0	1	2	3
0	136	194	198	201
1	194	135	194	196
2	201	194	135	200
3	202	197	198	135

- Lokaler Speicher langsamer als beim E5-4600
- Zugriffszeit weniger abhängig von der Entfernung im Netzwerk, maximale Entfernung ein QPI Link
- Maximaler Unterschied etwa Faktor 1.5

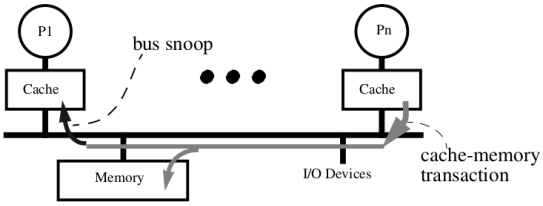
[1] `https://software.intel.com/en-us/blogs/2014/01/28/memory-latencies-on-intel-xeon-processor-e5-4600-and-e7-4800-pr`
[2] `https://software.intel.com/en-us/blogs/2014/01/28/memory-latencies-on-intel-xeon-processor-e5-4600-and-e7-4800-pr`

3.2 Cache Kohärenz

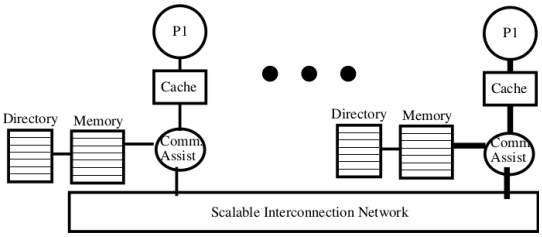


Protokolltypen

Snooping basierte Protokolle



Verzeichnis basierte Protokolle

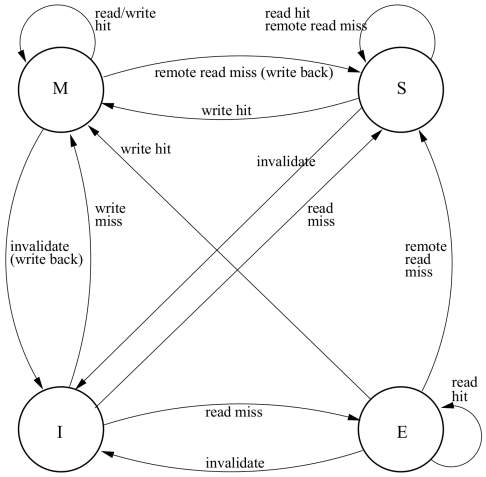


Bus Snooping, MESI

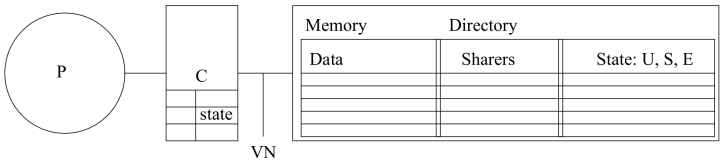
- Bus erlaubt einfaches, effizientes Protokoll zur Cache-Kohärenz.
- Beispiel MESI: Jeder Cache-Block hat einen der folgenden Zustände:

Zustand	Bedeutung
I	Eintrag ist nicht gültig
E	Eintrag gültig, Speicher aktuell, keine Kopien vorhanden
S	Eintrag gültig, Speicher aktuell, weitere Kopien vorhanden
M	Eintrag gültig, Speicher ungültig, keine Kopien vorhanden

- Erweitert write-back Protokoll um Cache-Kohärenz.
- Varianten: MOESI (AMD) und MESIF (neuere Intel-Prozessoren) ermöglichen direkte Weiterleitung von Daten aus Cache eines Prozessors/Kerns an einen anderen
- Cache-Controller hört Verkehr auf Bus ab (schnüffelt) und kann folgende Zustandübergänge vornehmen (aus Sicht *eines* Controllers):



Verzeichnisbasierte Cache-Kohärenz

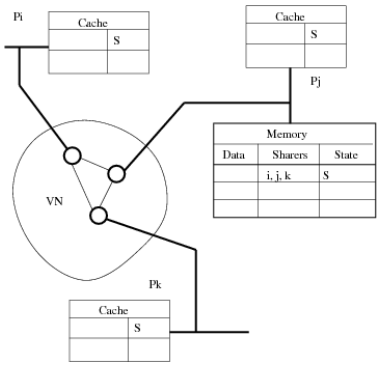


Zustände:

Cache–Block		Hauptspeicherblock	
Zust.	Beschreibung	Zust.	Beschreibung
I	Block ungültig	U	niemand hat den Block
S	$\geq$ 1 Kopien vorhanden, Caches und Speicher sind aktuell	S	wie links
E	genau einer hat den Block geschrieben (entspricht M in MESI)	E	wie links

Beispiel

Situation: Drei Prozessoren P_i, P_j, und P_k haben eine Cache Line im Zustand shared. Home Node dieses Speicherblockes ist P_j



Aktionen:

1. Prozessor P_i schreibt in die Cache-Line (write hit): Nachricht an Verzeichnis, dieses benachrichtet Caches von P_j und P_k, Nachfolgezustand ist E in P_i
2. Prozessor P_k liest von diesem Block (read miss): Verzeichnis holt Block von P_i, Verzeichnis schickt Block an P_k

Zustandsübergänge (aus Sicht des Verzeichnisses):

Z	Aktion	Folgt	Beschreibung
U	read miss	S	Block wird an Cache übertragen, Bitvektor enthält wer die Kopie hat.
	write miss	E	Block wird an den anfragenden Cache übertragen, Bitvektor enthält wer die gültige Kopie hat.
S	read miss	S	anfragender Cache bekommt Kopie aus dem Speicher und wird in Bitvektor vermerkt.
	w miss/hit	E	Anfrager bekommt (falls miss) eine Kopie des Speichers, Verzeichnis sendet invalidate an alle übrigen Besitzer einer Kopie.
E	read miss	S	Eigentümer des Blockes wird informiert, dieser sendet Block zurück zum Heimatknoten und geht in Zustand S, Verzeichnis sendet Block an anfragenden Cache.
	write back	U	Eigentümer will Cache–Block ersetzen, Daten werden zurückgeschrieben, niemand hat den Block.
	write miss	E	Der Eigentümer wechselt. Alter Eigentümer wird informiert und sendet Block an Heimatknoten, dieser sendet Block an neuen Eigentümer.

Problemfälle

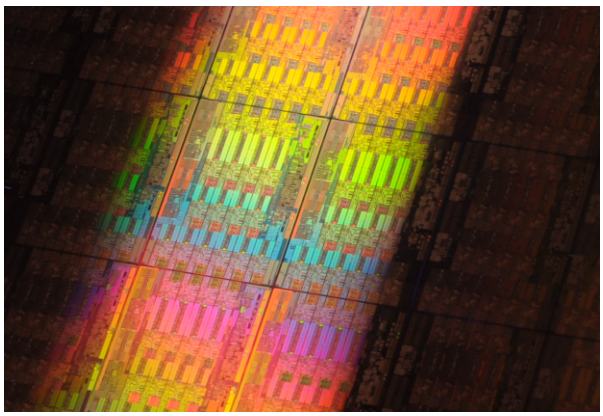
Probleme der ccNUMA-Architekturen (ccNUMA = cache coherent NUMA):

- false sharing: Zwei Prozessoren lesen und schreiben verschiedene Speicherstellen, die sich zufällig im selben Block befinden (Wahrscheinlichkeit steigt mit Blockgröße)
- capacity miss: Datenmenge die ein Prozessor bearbeitet (working set) passt nicht in den Cache und diese Daten sind im Hauptspeicher eines anderen Prozessors

Lösung für das Kapazitätsproblem: Cache Only Memory Architecture (COMA), Software Distributed Shared Memory. Seiten des Hauptspeichers (z. B. 4-16 KB) können automatisch migriert werden.

3.3 Beispiele

Intel Xeon Haswell-EP



- 2-Sockel System
- QPI-Interconnect
- bis zu 2 × 2.3 GHz 18-Kernprozessoren + 2-fach Hyperthreading
- bis 45 MB L3-Cache pro Prozessor
- bis 3 Terrabyte Hauptspeicher

Beispiel: SGI UV 1000

- cc-NUMA Shared-Memory Maschine
- besteht aus Blades mit je 2 Intel Xeon 2.27 GHz 8-Kern Prozessoren, 128 GB Speicher
- auf Blade QPI-Verbindung zu speziellem, schnellem NUMAlink5 Netzwerk, Fat-Tree Architektur
- bis zu 4'096 Kerne
- bis zu 32 TB Hauptspeicher, davon 16 TB für einen Job nutzbar
- größte Maschine: 37 TFlop Peak ("Blacklight", Pittsburgh Supercomputing Center, 2010), langsamste TOP 500 Maschine 2010: 57.5 TFlop Peak, 2015: 145.6 TFlop Peak

4 Parallele Programmiermodelle für Shared-Memory Maschinen: Grundlagen

4.1 Beispiel: Skalarprodukt

Skalarprodukt zweier Vektoren der Länge N:

$$s = x \cdot y = \sum_{i=0}^{N-1} x_i y_i.$$

Parallelisierungsidee:

1. Summanden $x_i y_i$ unabhängig
2. $N \geq P$, bilde $I_p \subseteq \{0, \ldots, N-1\}$, $I_p \cap I_q = \emptyset \; \forall p \neq q$
   Jeder Prozessor berechnet dann die Teilsumme $s_p = \sum_{i \in I_p} x_i y_i$
3. Summation der Teilsummen z.B. für $P = 8$:

$$s = \underbrace{\underbrace{\underbrace{s_0 + s_1}_{s_{01}} + \underbrace{s_2 + s_3}_{s_{23}}}_{s_{0123}} + \underbrace{\underbrace{s_4 + s_5}_{s_{45}} + \underbrace{s_6 + s_7}_{s_{67}}}_{s_{4567}}}_{s}$$

Grundlegende Begriffsbildungen

Sequentielles Programm: Folge von Anweisungen die der Reihe nach abgearbeitet werden.

Sequentieller Prozess: Aktive Ausführung eines sequentiellen Programmes.

Parallele Berechnung: Menge interagierender sequentieller Prozesse.

Paralleles Programm: Beschreibt parallele Berechnung. Gegeben durch Menge von sequentiellen Programmen.

Notation für parallele Programme

- Möglichst einfach, losgelöst von praktischen Details
- Erlaubt verschiedene Programmiermodelle

Programm 4.1 (Muster eines parallelen Programmes)**.**

```
parallel <Programmname>
{
    // Sektion mit globalen Variablen (von allen Prozessen zugreifbar)
    process <Prozessname-1> [<Kopienparameter>]
    {
        // lokale Variablen, die nur von Prozess <Prozessname-1>
        // gelesen und geschrieben werden können
        // Anwendungen in C-ähnlicher Syntax. Mathematische
        // Formeln oder Prosa zur Vereinfachung erlaubt.
    }
    ...
    process <Prozessname-n> [<Kopienparameter>]
    {
        ...
    }
}
```

Variablendeklaration

double x, $y[P]$;

Initialisieren

int $n[P] = \{1[P]\}$;

Skalarprodukt mit zwei Prozessen

Programm 4.2 (Skalarprodukt mit zwei Prozessen)**.**

```
parallel two-process-scalar-product
{
    const int N=8;          // Problemgröße
    double x[N], y[N], s=0;// Vektoren, Resultat
    process Π1
    {
        int i;
        double ls=0;
        for (i = 0; i < N/2; i++)
            ls += x[i]*y[i];
        s=s+ls;             // Gefahr!
    }
    process Π2
    {
        int i;
        double ls=0;
        for (i = N/2; i < N; i++)
```

```
        ls += x[i]*y[i];
      s=s+ls;                    // Gefahr!
    }
}
```

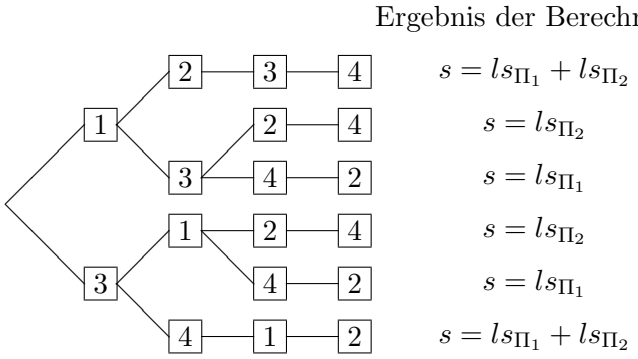
- Variablen sind global, jeder Prozess bearbeitet Teil der Indizes
- Kollision bei schreibendem Zugriff!

4.2 Kritischer Abschnitt

- Hochsprachenanweisung $s = s + ls$ wird zerlegt in Maschinenbefehle:

	Prozess Π_1		Prozess Π_2
1	lade s in R1	3	lade s in R1
	lade ls in R2		lade ls in R2
	add R1 und R2, Ergebnis in R3		add R1 und R2, Ergebnis in R3
2	speichere R3 nach s	4	speichere R3 nach s

- Die Ausführungsreihenfolge der Anweisungen verschiedener Prozesse ist nicht festgelegt.
- Mögliche Ausführungsreihenfolgen sind:



- Nur die Reihenfolgen 1-2-3-4 oder 3-4-1-2 sind korrekt.

Was ist ein kritischer Abschnitt?
Wir betrachten folgende Situation:

- Anwendung besteht aus P nebenläufigen Prozessen, diese werden also zeitgleich bearbeitet
- von einem Prozeß ausgeführte Anweisungen werden in zusammenhängende Gruppen eingeteilt

  - kritische Abschnitte
  - unkritische Abschnitte

- kritischer Abschnitt: Folge von Anweisungen, welche lesend oder schreibend auf *gemeinsame Variable* zugreift.
- Anweisungen eines kritischen Abschnitts dürfen nicht gleichzeitig von zwei oder mehr Prozessen bearbeitet werden

$\rightarrow$ man sagt nur ein Prozeß darf sich im kritischen Abschnitt befinden

- Wir notieren dies *zunächst* mit eckigen Klammern

$$[\ \langle \text{Anweisung } 1 \rangle; \ldots; \langle \text{Anweisung } n \rangle;\ ]$$

- Das Symbol „[“ wählt einen Prozess aus der den kritischen Abschnitt bearbeiten darf, alle anderen warten.

- Effiziente Umsetzung erfordert Hardwareinstruktionen.

4.2.1 Wechselseitiger Ausschluß

2 Arten der Synchronisation sind unterscheidbar

- Bedingungssynchronisation
- Wechselseitiger Ausschluss

Wechselseitiger Ausschluss besteht aus *Eingangsprotokoll* und *Ausgangsprotokoll*:

Programm 4.3 (Einführung wechselseitiger Ausschluss)**.**

```
parallel critical-section
{
    process Π [int p ∈ {0, ..., P − 1}]
    {
        while (1)
        {
            Eingangsprotokoll;
            kritischer Abschnitt;
            Ausgangsprotokoll;
            unkritischer Abschnitt;
        }
    }
}
```

Folgende Kriterien sollen erfüllt werden:

1. *Wechselseitiger Ausschluss.* Höchstens ein Prozess führt den kritischen Abschnitt zu einer Zeit aus.

2. *Verklemmungsfreiheit.* Falls zwei oder mehr Prozesse versuchen in den kritischen Abschnitt einzutreten muss es genau einer nach endlicher Zeit auch schaffen.

3. *Keine unnötige Verzögerung.* Will ein Prozess in den kritischen Abschnitt eintreten während alle anderen ihre unkritischen Abschnitte bearbeiten so darf er nicht am Eintreten gehindert werden.

4. *Schließliches Eintreten.* Will ein Prozess in den kritischen Abschnitt eintreten so muss er dies nach endlicher Wartezeit auch dürfen (dazu nimmt man an, daß jeder den kritischen Abschnitt auch wieder verlässt).

Petersons Algorithmus: Eine Softwarelösung

Wir betrachten zunächst nur zwei Prozesse und entwickeln die Lösung schrittweise ...

Erster Versuch: Warte bis der andere *nicht* drin ist

int *in1*=0, *in2*=0; // 1=drin

Π_1:	Π_2:
while (*in2*) ;	**while** (*in1*) ;
in1=1;	*in2*=1;
krit. Abschnitt;	krit. Abschnitt;

- Es werden keine Maschinenbefehle benötigt
- Problem: Lesen und Schreiben ist nicht atomar

Petersons Algorithmus: Zweite Variante

Erst besetzen, dann testen

int *in1*=0, *in2*=0;

Π_1:	Π_2:
in1=1;	*in2*=1;
while (*in2*) ;	**while** (*in1*) ;
krit. Abschnitt;	krit. Abschnitt;

Problem: Verklemmung möglich

Petersons Algorithmus: Dritte Variante

Löse Verklemmung durch Auswahl eines Prozesses

Programm 4.4 (Petersons Algorithmus für zwei Prozesse)**.**

```
parallel Peterson-2
{
    int in1=0, in2=0, last=1;

    process Π1                          process Π2
    {                                   {
        while (1) {                         while (1) {
            in1=1;                              in2=1;
            last=1;                             last=2;
            while (in2 ∧ last==1) ;             while (in1 ∧ last==2) ;
            krit. Abschnitt;                    krit. Abschnitt;
            in1=0;                              in2=0;
            unkrit. Abschnitt;                  unkrit. Abschnitt;
        }                                   }
    }                                   }
}
```

Konsistenzmodelle

Bisherige Beispiele basieren auf dem Prinzip der *Sequentiellen Konsistenz*:

1. *Lese- und Schreiboperationen werden in Programmordnung beendet*
2. *diese Reihenfolge ist für alle Prozessoren konsistent sichtbar*

Hier erwartet man, dass $a = 1$ gedruckt wird:

```
int a = 0,flag=0;          // wichtig!
process Π1                 process Π2
    ...                        ...
    a = 1;
    flag=1;                    while (flag==0)  ;
                               print a ;
```

Hier erwartet man, dass nur für einen die **if**-Bedingung wahr wird:

```
int a = 0, b = 0;          // wichtig!
process Π1                 process Π2
    ...                        ...
    a = 1;                     b = 1;
    if (b == 0) ...            if (a == 0) ...
```

Warum keine sequentielle Konsistenz?

- *Umordnen von Anweisungen*: Optimierende Compiler können Operationen aus Effizienzgründen umordnen. Das erste Beispiel funktioniert dann nicht mehr!
- *Out-of-order execution*: z. B. Lesezugriffe sollen langsame Schreibzugriffe (invalidate) überholen können (solange es nicht die selbe Speicherstelle ist). Das zweite Beispiel funktioniert dann nicht mehr!

Total store ordering: Lesezugriff darf Schreibzugriff überholen

Weak consistency: Alle Zugriffe dürfen einander überholen

Reihenfolge kann durch spezielle Maschinenbefehle erzwungen werden, z. B. *fence*: alle Speicherzugriffe beenden bevor ein neuer gestartet wird.

Diese Operationen werden eingefügt,

- bei annotieren von Variablen („Synchronisationsvariable“)
- bei parallelen Anweisungen (z. B. FORALL in HPF)
- vom Programmierer der Synchronisationsprimitive (z. B. Semaphore)

Peterson für P Prozesse

Idee: Jeder durchläuft $P-1$ Stufen, Jeweils der letzte auf einer Stufe ankommende muss warten

Variablen:

- $in[i]$: Stufe $\in \{1,\ldots,P-1\}$ (!), die Π_i erreicht hat
- $last[j]$: Nummer des Prozesses der zuletzt auf Stufe j ankam

Programm 4.5 (Petersons Algorithmus für P Prozesse)**.**

```
parallel Peterson-P
{
    const int P=8;
    int in[P] = {0[P]};
    int last[P] = {0[P]};
    ...
}
```

Programm 4.6 (Petersons Algorithmus für P Prozesse cont.)**.**

```
parallel Peterson-P cont.
{
    process Π [int i ∈ {0, ..., P − 1}]
    {
        int j,k;
```

```
        while (1)
        {
            for (j=1; j ≤ P − 1; j++)          // durchlaufe Stufen
            {
                in[i] = j;                      // ich bin auf Stufe j
                last[j] = i;                    // ich bin der letzte auf Stufe j
                for (k = 0; k < P; k++)         // teste alle anderen
                    if (k ≠ i)
                        while (in[k]>=in[i] ∧ last[j]==i) ;
            }
            kritischer Abschnitt;
            in[i] = 0;                          // Ausgangsprotokoll
            unkritischer Abschnitt;
        }
    }
}
```

- $O(P^2)$ Tests notwendig für Eintritt
- Strategie ist fair, wer als erstes kommt darf als erster rein

4.2.2 Hardware Locks

Hardwareoperationen zur Realisierung des wechselseitigen Ausschlusses:

- *test-and-set*: überprüfe ob eine Speicherstelle den Wert 0 hat, wenn ja setze die Speicherstelle auf eins (als unteilbare Operation).
- *fetch-and-increment*: Hole den Inhalt einer Speicherstelle in ein Register und erhöhe den Inhalt der Speicherstelle um eins (als unteilbare Operation).
- *atomic-swap*: Vertausche den Inhalt eines Registers, also eins, mit dem Inhalt einer Speicherstelle in einer unteilbaren Operation.

In allen Maschinenbefehlen muß ein Lesezugriff gefolgt von einem Schreibzugriff ohne Unterbrechung durchgeführt werden!

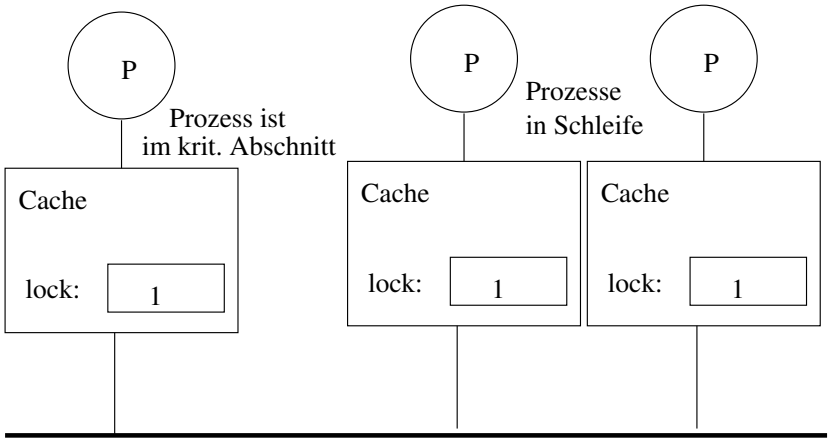
Ziel: Maschinenbefehl und Cache-Kohärenz Modell stellt alleiniges Eintreten in den kritischen Abschnitt sicher und erzeugt geringen Verkehr auf dem Verbindungsnetzwerk

Programm 4.7 (Spin Lock)**.**

```
parallel spin–lock
{
    const int P = 8;          // Anzahl Prozesse
    int lock=0;               // Variable zur Absicherung

    process Π [int p ∈ {0, ..., P − 1}]
    {
```

```
        ...
        while ( atomic − swap(& lock))  ;
        ...                          // kritischer Abschnitt
        lock = 0;
        ...
    }
}
```

Was passiert im System?



Die beiden wartenden Prozesse erzeugen hohen Busverkehr
Ablauf bei MESI Protokoll:Variable *lock* ist 0 und ist in keinem der Caches

- Prozess Π_0 führt die $atomic - swap$–Operation aus

→ Lesezugriff führt zu read miss, Block wird aus dem Speicher geholt und bekommt den Zustand E (wir legen MESI zugrunde).

→ Nachfolgendes Schreiben ohne weiteren Buszugriff, Zustand von E nach M.

- anderer Prozess Π_1 führt $atomic - swap$–Operation aus

→ Read miss führt zum Zurückschreiben des Blockes durch Π_0, der Zustand beider Kopien ist nun, nach dem Lesezugriff, S.

- Schreibzugriff von Π_1 invalidiert Kopie von Π_0 und Zustand in Π_1 ist M.
- $atomic - swap$ gibt 1 in Π_1 und kritischer Abschnitt wird von Π_1 nicht betreten.
- Führen beide Prozesse die $atomic - swap$–Operation gleichzeitig aus entscheidet letztendlich der Bus wer gewinnt.
- Angenommen Cache C_0 von Prozessor P_0 als auch C_1 haben je eine Kopie des Cache–Blockes im Zustand S vor Ausführung der $atomic - swap$–Operation
- Beide lesen zunächst den Wert 0 aus der Variable *lock*.

- Beim nachfolgenden Schreibzugriff konkurrieren beide um den Bus um eine invalidate Meldung zu platzieren.
- Der Gewinner setzt seine Kopie in Zustand M, Verlierer seine in den Zustand I. Die Cache–Logik des Verlierers findet beim Schreiben den Zustand I vor und muss den $atomic - swap$–Befehl veranlassen doch den Wert 1 zurückzuliefern (der atomic-swap–Befehl ist zu dieser Zeit ja noch nicht beendet).

Verbessertes Lock

Idee: Mache keinen Schreibzugriff solange der kritische Abschnitt besetzt ist

Programm 4.8 (Verbesserter Spin Lock)**.**

```
parallel improved–spin–lock
{
    const int P = 8;          // Anzahl Prozesse
    int lock=0;               // Variable zur Absicherung

    process Π [int p ∈ {0, ..., P − 1}] {
        ...
        while (1)
            if (lock==0)
                if (test − and − set(& lock)==0 )
                    break;
        ...                   // kritischer Abschnitt
        lock = 0;
        ...
    }
}
```

Verbessertes Lock

1. Problem: Strategie gewährleistet keine Fairness
2. Situation mit drei Prozessen: Zwei wechseln sich immer ab, der dritte kommt nie rein
3. Aufwand falls P Prozesse gleichzeitg eintreten wollen: $O(P^2)$, Zuweisung $lock = 0$ bedingt P Bustransaktionen für Cache Block Kopien
4. Lösung ist ein Queuing Lock: Bei Austritt aus dem kritischen Abschnitt wählt der Prozess seinen Nachfolger aus

Ticketalgorithmus:

- Fairness mit Hardware-Lock
- Idee: Vor dem Anstellen an der Schlange zieht man eine Nummer. Der mit der kleinsten Nummer kommt als nächster dran.

4.2.3 Ticketalgorithmus

Programm 4.9 (Ticket Algorithmus für P Prozesse)**.**

```
parallel Ticket
{
    const int P=8;
    int number=0;
    int next=0;

    process Π [int i ∈ {0, ..., P − 1}]
    {
        int mynumber;
        while (1)
        {
            [mynumber=number; number=number+1;]
            while (mynumber≠next) ;
            kritischer Abschnitt;
            next = next+1;
            unkritischer Abschnitt;
        }
    }
}
```

1. Fairness basiert darauf, daß Ziehen der Zahl nicht lange dauert. Möglichkeit einer Kollision ist kurz.

2. Geht auch bei Überlauf der Zähler, (solange `MAXINT>P`)

3. Inkrementieren von *next* geht ohne Synchronisation, da dies immer nur einer machen kann

Bedingter kritischer Abschnitt

- Erzeuger-Verbraucher Problem:
  - m Prozesse P_i (Erzeuger) erzeugen Aufträge, die von n anderen Prozessen C_j (Verbraucher) abgearbeitet werden sollen.
  - Die Prozesse kommunizieren über eine zentrale Warteschlange (WS) mit k Plätzen.
  - Ist die WS voll müssen die Erzeuger warten, ist die WS leer müssen die Verbraucher warten.

- Problem: Wartende dürfen den (exklusiven) Zugriff auf die WS nicht blockieren!

- Kritischer Abschnitt (Manipulation der WS) darf nur Betreten werden *wenn* WS nicht voll (für Erzeuger), bzw. nicht leer (für Verbraucher) ist.

- Idee: Probeweises Betreten und busy-wait:

Programm 4.10 (Erzeuger–Verbraucher Problem mit aktivem Warten)**.**

```
parallel producer-consumer-busy-wait
{
    const int m = 8; n = 6; k = 10;
    int orders=0;
    process P [int 0 ≤ i < m]
    {
        while (1) {
            produziere Auftrag;
            CSenter;
            while (orders==k){
                CSexit;
                CSenter;
            }
            speichere Auftrag;
            orders=orders+1;
            CSexit;

        }
    }
    process C [int 0 ≤ j < n]
    {
        while (1) {

            CSenter;
            while (orders==0){
                CSexit;
                CSenter;
            }
            lese Auftrag;
            orders=orders-1;
            CSexit;
            bearbeite Auftrag;
        }
    }
}
```

- Ständiges Betreten und Verlassen des kritischen Abschnittes ist ineffizient wenn mehrere Warten (Trick vom verbesserten Lock hilft nicht)
- (Praktische) Abhilfe: Zufällige Verzögerung zwischen *CSenter*/*CSexit*, *exponential back-off*.

4.3 Parametrisierung von Prozessen

- Prozesse enthalten teils identischen Code (auf unterschiedlichen Daten)
- Parametrisiere den Code mit einer Prozessnummer, wähle zu bearbeitende Daten mit dieser Nummer aus
- SPMD = single program multiple data

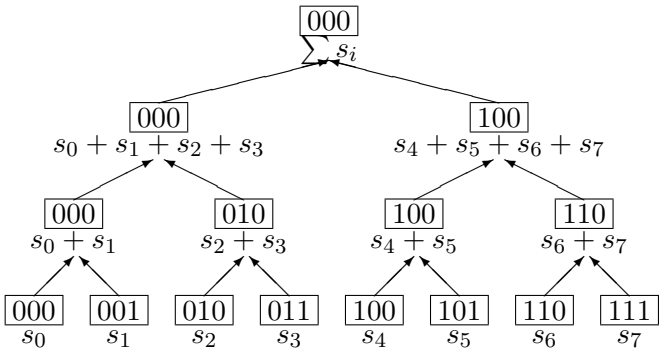
Programm 4.11 (Skalarprodukt mit P Prozessoren)**.**

```
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 ∈ {0, ..., P − 1}]
    {
        int i; double ss = 0;
        for (i = N ∗ p/P; i < N ∗ (p + 1)/P; i++)
            ls += x[i]*y[i];
        [s = s + ls];         // Hier warten dann doch wieder alle
    }
}
```

$[s = s + ls]$ wird sequentiell abgearbeitet!

Kommunikation in hierarchischer Struktur

Baumartige Organisation der Kommunikationsabfolge mit $\mathrm{ld}\, P$ Stufen



In Stufe $i = 0, 1, \ldots$

- Prozesse, deren letzte $i + 1$ Bits 0 sind, holen
- Ergebnisse von den Prozessoren deren letzte i Bits 0 und deren Bit i 1 ist

Parallelisierung der Summe

Programm 4.12 (Parallele Summation)**.**

```
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 letzte 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];
            }
        }
    }
}
```

- Neue globale Variablen: $s[P]$ Teilergebnisse $flag[P]$ zeigt an, dass Prozessor fertig ist
- Das Warten nennt man *Bedingungssynchronisation*
- In diesem Beispiel konnte wechselseitiger Ausschluß durch eine Bedingungssynchronisation ersetzt werden. Dies geht nicht immer!
- Liegt daran, dass wir hier die Reihenfolge vorab festgelegt haben

Lokalisieren

Ziel: Vermeiden globaler Variablen

Programm 4.13 (Skalarprodukt mit lokalen Daten)**.**

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

    process Π [ int p ∈ {0, ..., P − 1}]
    {
        double x[N/P], y[N/P]; // Annahme N durch P teilbar
                               // Lokaler Ausschnitt der Vektoren
        int i;
        double ls=0;

        for (i = 0,i < N/P;i++)
            ls = ls + x[i] ∗ y[i];
        [s = s + ls;]
    }
}
```

Jeder speichert nur N/P Indizes (einer mehr falls nicht exakt teilbar), *diese beginnen immer mit der lokalen Nummer 0*

Jeder lokale Index entspricht einem globalen Index im sequentiellen Programm:

$$i_{\text{global}}(p) = i_{\text{lokal}} + p * N/P$$

5 OpenMP

5.1 Grundlagen

- OpenMP ist eine spezielle Implementierung von Multithreading.
- Die aktuelle Version ist 4.5 (November 2015)
- sowohl für Fortran als auch für C/C++ verfügbar
- funktioniert mit verschiedenen Betriebssystemen (e.g. Linux, Windows, MacOS, Solaris)
- in verschiedene Compiler integriert (e.g. Intel ICC $\geq$ 8.0, GCC $\geq$ 4.2, Visual Studio $\geq$ 2005, Clang $\geq$ 3.7, ...)
- GCC ab 4.2 unterstützt OpenMP 2.5, ab 4.4 OpenMP 3.0, ab 4.7 OpenMP 3.1, ab 4.9 OpenMP 4.0 (ohne Akzelerator-Unterstützung), ab 6.1 OpenMP 4.5 voll unterstützt

Thread-Model von OpenMP

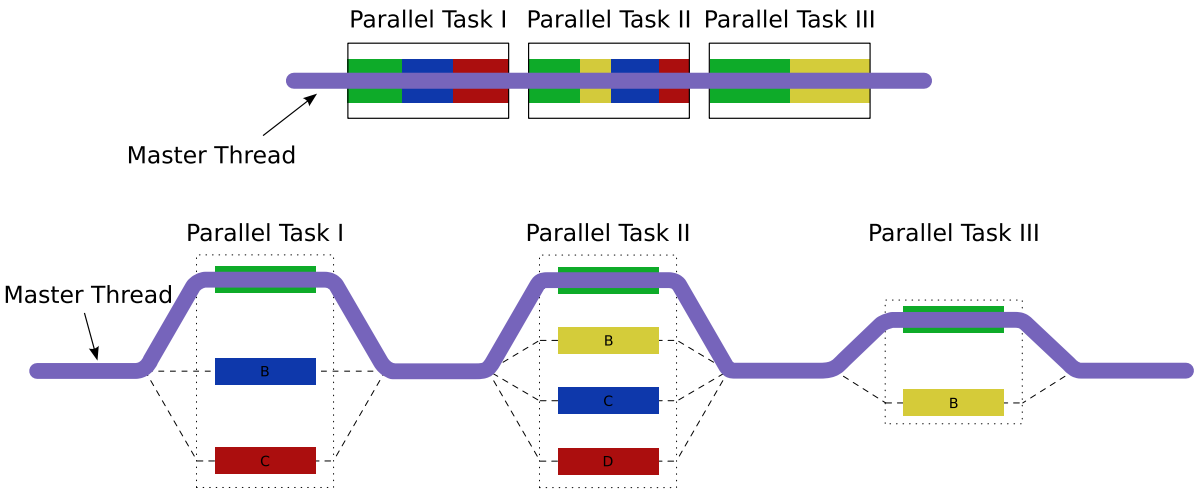


figure from Wikipedia: "OpenMP"

OpenMP Anweisungen

OpenMP directives

- sind eine Erweiterung der jeweiligen Sprache
- bestehen aus speziellen Anweisungen an den Präprozessor im Sourcecode, die von einem geeigneten Compiler ausgewertet werden
- beginnen immer mit `#pragma omp` gefolgt von einem Schlüsselwort und optional von Argumenten

OpenMP Bestandteile

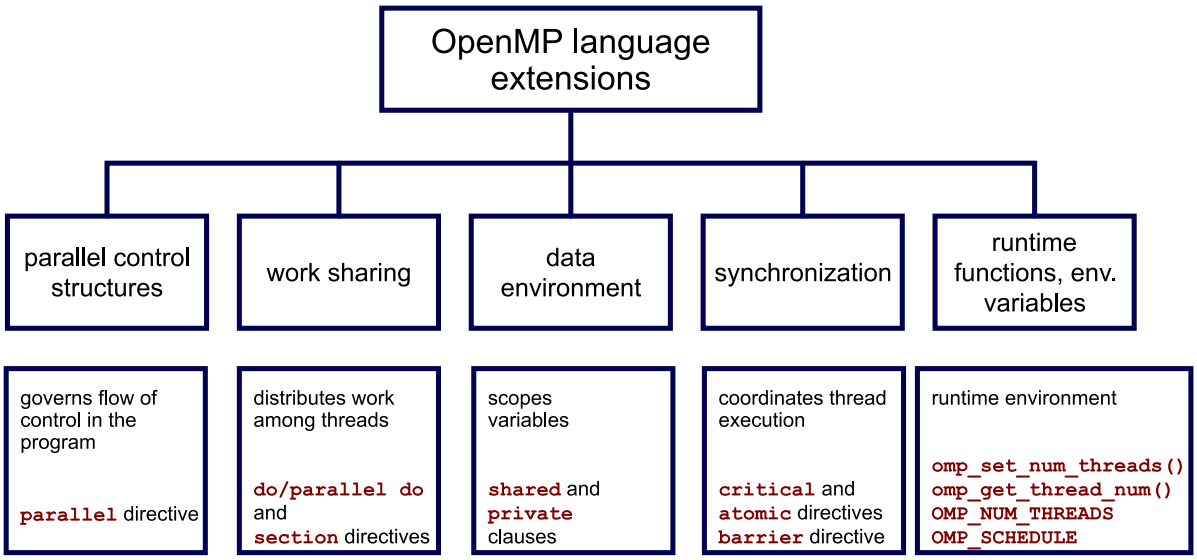


figure from Wikipedia: "OpenMP"

+ SIMD, + Beschleunigerkarten

Skalarprodukt mit OpenMP

```
double ScalarProduct(std::vector<double> a,
                     std::vector<double> b)
{
  const int N=a.size();
  int i;
  double sum = 0.0;
  #pragma omp parallel for
    for (i=0;i<N;++i)
      sum += a[i] * b[i];
  return sum;
}
```

5.2 Parallelisierung

Thread-Initialisierung

parallel startet einen Block der parallel ausgeführt wird

for Die Wiederholungen werden unter den Threads aufgeteilt (wird oft mit `parallel` zu einer Anweisung `#pragma omp parallel for` kombiniert).

sections innerhalb eines `sections` Blocks gibt es mehrere `section` Blöcke, die voneinander unabhängig sind und gleichzeitig ausgeführt werden können.

task der Inhalt des nachfolgenden Blocks wird als `task` erzeugt und (bei Gelegenheit) durch einen Thread abgearbeitet.

Alle Variante ein paralleles Programm zu starten haben zusätzliche Optionen, z.B. eine List mit Zugriffsregelung für Variablen oder

if wenn bei der Thread-Initialisierung eine `if` Anweisung steht (innerhalb des Pragmas), wird der Block nur dann parallel ausgeführt, wenn die Bedingung nach dem `if` wahr ist.

Zugriff auf Variablen

shared Alle Threads verwenden die gleichen (geteilten) Daten.

private Jeder Thread hat eine lokale Version der Variablen. Diese wird nicht initialisiert.

firstprivate Jeder Thread hat eine lokale Version der Variablen. Diese wird mit dem Inhalt der entsprechenden Variablen aus dem Aufführenden Thread initialisiert.

lastprivate Jeder Thread hat eine lokale Version der Variablen. Diese wird nicht initialisiert. Am Ende des parallelen Blocks wird der Inhalt des letztes Threads in die Variable des aufrufenden Threads kopiert.

default legt das Standardverhalten für Variablen fest. Ist entweder `shared`, `private`, `firstprivate` oder `none`.

reduction Durch die Anweisung `reduction(`*`operator`*`,`*`variable`*`)` erhält jeder Thread eine lokale Version von *`variable`* aber am Ende werden alle lokalen Kopien mit dem Operator *`operator`* verknüpft. Dies erfolgt so effizient wie möglich. Mögliche Operatoren sind `+ * - / & ^ |`. Seit OpenMP 4.0 können auch eigene Operatoren definiert werden. Die lokalen Variablen werden mit einer der Operation entsprechenden Variante initialisiert.

```
int main()
{
  int section_count = 0;
  #pragma omp parallel
  #pragma omp sections firstprivate(section_count)
  {
    #pragma omp section
    {
      section_count++;
      printf("section_count␣%d\n", section_count);
    }
    #pragma omp section
    {
      printf("section_count␣%d\n", section_count);
    }
  }
  return 1;
}
```

`threadprivate` Variablen

- Für jeden Thread wird eine lokale Version einer globalen Variablen oder statischen Variablen angelegt.
- Für die Variable wird die bei Ihrer ursprünglichen Definition definierte Initialisierung verwendet.
- Wann genau die Variable erzeugt wird, oder wann sie wieder zerstört wird, ist nicht festgelegt. Es erfolgt vor der ersten Verwendung bzw. nach Verlassen des parallelen Blocks
- Unter Umständen bleiben `threadprivate` Variablen bis zum nächsten Block erhalten (wenn die Anzahl Threads gleich bleibt, parallele Regionen nicht verschachtelt sind ...).

Globale Variable:

```
int counter = 0;
#pragma omp threadprivate(counter)
int increment_counter()
{
  counter++;
  return(counter);
}
```

Statische Variable:

```
int increment_counter_2()
{
  static int counter = 0;
  #pragma omp threadprivate(counter)
  counter++;
  return(counter);
}
```

```
#include <omp.h>

int x, y, z[1000];
#pragma omp threadprivate(x)

int main(int a) {
  const int c = 1;
  int i = 0;
  #pragma omp parallel default(none) private(a) shared(z)
  {
    int j = omp_get_num_threads();
    // O.K.  - j is declared within parallel region
    a = z[j]; // O.K.  - a is listed in private clause
              // - z is listed in shared clause
    x = c; // O.K.  - x is threadprivate, c has const-qualified
       type
```

```
    z[i] = y; // Error - cannot reference i or y here
17  #pragma omp for firstprivate(y)
    // Error - Cannot reference y in the firstprivate clause
19  for (i = 0; i < 10; i++) {
      z[i] = i; // O.K. - i is the loop iteration variable
21  }
    z[i] = y; // Error - cannot reference i or y here
23 }
}
```

Synchronisierung

critical wechselseitiger Ausschluss: der Block wird immer nur von einem Thread auf einmal ausgeführt.

atomic ermöglicht Zugriff atomaren Zugriff auf eine Speicherzelle, wenn möglich mit Hardwareanweisungen. Es wird unterschieden zwischen `read`, `write`, `update` und `capture`.

single der Inhalt des `single` Blocks wird nur von einem einzigen Thread ausgeführt. Die anderen Threads warten am Ende des Blocks.

master der Inhalt des `master` Blocks wird nur vom Master-Thread ausgeführt. Er wird von allen anderen Threads übersprungen.

barrier jeder Thread wartet bis alle anderen Threads an der Barriere angekommen sind.

flush schreibt alle Variablen in der nachfolgenden Liste in den Speicher.

nowait normalerweise wartet ein Thread am Ende eines Blocks. Wird `nowait` verwendet, dann macht er sofort mit den nächsten Anweisungen außerhalb des Blocks weiter.

Atomic

Eine Anweisung, die atomar ausgeführt werden kann, hat die Form `x Operator Ausdruck`. Mögliche Operatoren sind `+= *= -= /= &= ^= |= <<= >>=`. Dabei wird der Ausdruck auf der rechten Seite erst ausgewertet (dies erfolgt nicht atomar) und dann wird die Speicherzelle auf der linken Seite in einer atomaren Anweisung mit dem Ergebnis aktualisiert. Außerdem gibt es die atomaren Anweisungen `x++, ++x, x++, --x`. Je nach Laufzeitsystem kann eine atomare Anweisung effizienter ausgeführt werden als bei einer Realisierung mit einer critical section.

```
void atomic_example(float *x, float *y, int *index, int n)
2 {
  int i;
4 #pragma omp parallel for shared(x, y, index, n)
  for (i=0; i<n; i++) {
6   #pragma omp atomic update
    x[index[i]] += work1(i);
8   y[i] += work2(i);
  }
10 }
```

Lastverteilung für parallele For-Schleifen

Die Aufteilung der Iterationen auf die einzelnen Threads kann durch die Option `schedule` des Pragmas `for` gesteuert werden. Jeder Thread bekommt dabei Indexblöcke (chunks) einer vorgegebenen Größe . Nach `schedule` folgt einer von fünf möglichen Parameter und optional eine sogenannte Blockgröße (chunk size). Die Blockgröße ist ein Integerwert, der zur Zeit der Übersetzung bekannt sein muss. Der Standardwert für `schedule` ist abhängig vom konkreten Compiler.

static die chunks werden zu Beginn der Schleife auf die verschiedenen Threads verteilt.

dynamic immer wenn ein Thread einen chunk abgearbeitet hat bekommt er einen neuen.

guided die Blockgröße wird proportional zur Anzahl noch abzuarbeitender Iterationen geteilt durch die Anzahl Threads gewählt (d.h. die Blockgröße wird mit der Zeit kleiner).

runtime die Lastverteilung wird durch die Laufzeitvariable `OMP_SCHEDULE` festgelegt. Die Angabe einer Blockgröße ist dann nicht erlaubt.

auto die Lastverteilung wird vom Compiler und/oder der Laufzeitumgebung übernommen.

```
#define CHUNK_SIZE 10
#pragma omp parallel for schedule(dynamic,CHUNK_SIZE)
```

Verbessertes Skalarprodukt mit OpenMP

```
double ScalarProduct(std::vector<double> a,
                     std::vector<double> b)
{
  const int N=a.size();
  int i;
  double sum = 0.0, temp;
  #pragma omp parallel for shared(a,b,sum) private(temp)
    for (i=0;i<N;++i)
    {
      temp = a[i] * b[i];
      #pragma omp atomic
      sum += temp;
    }
  return sum;
}
```

Weiter verbessertes Skalarprodukt mit OpenMP

```
double ScalarProduct(std::vector<double> a,
                     std::vector<double> b)
{
  const int N=a.size();
  int i;
```

```
  double sum = 0.0;
7 #pragma omp parallel for shared(a,b) reduction(+:sum)
    for (i=0;i<N;++i)
9     sum += a[i] * b[i];
  return(sum);
11 }
```

Parallele Ausführung verschiedener Aufgaben

```
1 #pragma omp parallel sections
  {
3   #pragma omp section
    {
5     A();
      B();
7   }
    #pragma omp section
9   {
      C();
11    D();
    }
13  #pragma omp section
    {
15    E();
      F();
17  }
  }
```

5.2.1 Tasks

Tasks

Eine Alternative zur Parallelisierung von Schleifen oder zur parallelen Ausführung von bestimmten Programmteilen ist die Erzeugung einer größeren Menge unabhängiger Aufgaben (Tasks), die unabhängig voneinander abgearbeitet werden können.

task der Block nach dem `task` pragma wird in einen Pool von Tasks eingereiht.

taskwait bei diesem Kommando wartet ein Thread auf alle Kinder, die seit seiner eigenen Entstehung gestartet wurden (aber nicht auf von diesen gestartete Tasks).

taskgroup der Thread wartet auf die Beendigung aller Kinder dieses Tasks und aller wiederum von diesen gestarteten Tasks.

taskyield der aktuelle Task kann pausiert werden und der Thread kann inzwischen einen anderen Task bearbeiten (z.B. weil der Task auf die Freigabe eines Locks wartet).

Taskyield

```
#include <omp.h>
void something_useful(void);
void something_critical(void);
void foo(omp_lock_t *lock, int n)
{
  int i;
  for (i=0; i<n; i++)
  #pragma omp task
  {
    something_useful();
    while ( !omp_test_lock(lock) )
    {
      #pragma omp taskyield
    }
    something_critical();
    omp_unset_lock(lock);
  }
}
```

Beendigung von Tasks

Manchmal soll die Bearbeitung eines parallelen Abschnitts vorzeitig beendet werden, z.B. weil ein Fehler aufgetreten ist, oder das Ergebnis fest stetht. Davon müssen alle anderen Threads benachrichtigt werden.

cancel Beendet die Prozesse der innersten Umgebung mit dem angegebenen Typ (entweder `parallel`, `sections`, `for` oder `taskgroup`. Die anderen Tasks/Chunks werden fertig bearbeitet, sofern sie vorher nicht einen `cancellation point` erreichen.

cancellation point markiert einen Punkt an dem der Thread prüft, ob ein anderer Thread `cancel` aufgerufen hat.

```
#include<exception>

int main()
{
  std::exception *ex = nullptr;
  #pragma omp parallel shared(ex)
  {
    #pragma omp for
    for (int i = 0; i < 1000; i++)
    {
      //no 'if' that prevents compiler optimizations
      try {
        causes_an_exception();
      }
      catch(std::exception *e) {
```

```
16          //still must remember exception for later handling
            #pragma omp atomic write
18          ex = e;
            #pragma omp cancel for
20        }
          //cancel worksharing construct
22      }

        //if an exception has been raised, cancel parallel region
2       if (ex) {
          #pragma omp cancel parallel
4       }
        phase_1();
6       #pragma omp barrier
        phase_2();
8     }
      //continue here if an exception has been thrown in the
         worksharing loop
10    if (ex) {
        //handle exception stored in ex
12    }
  }
```

5.2.2 SIMD

Seit Version 4.0 enthält OpenMP auch Kommandos mit denen sich die Verwendung der SIMD-Einheiten steuern lässt.

simd Dieses Pragma weist daraufhin, dass die folgende Schleife vektorisiert werden soll. Es hat folgende Optionen:

- **safelen** maximaler Abstand von zwei Iterationsindizes, die noch parallel ausgeführt werden dürfen (um Abhängigkeiten zu verhindern).
- **aligned** das Objekt in Klammern ist auf die Anzahl angegebener Bytes aligned.
- **linear** die Variablen in Klammer hängen linear vom Schleifenzähler ab und sind sonst wie private Variablen zu behandeln.

Außerdem gibt es die üblichen Optionen zur Regelung des Zugriffs auf Variablen.

declare simd steht vor einer Funktion und gibt an, dass die Funktion innerhalb einer vektorisierten Schleife verwendet werden darf und keine Funktionalitäten enthält, die sich nicht mit SIMD vertragen. Es gibt eine Reihe Optionen (siehe OpenMP Spezifikation).

5.3 OpenMP-Funktionen

Es gibt einige spezielle Funktionen, die in `omp.h` definiert sind, z.B.

- `int omp_get_num_procs();` liefert Anzahl verfügbarer Prozessoren
- `int omp_get_num_threads();` liefert Anzahl gestarteter Threads

- `int omp_get_thread_num();` liefert Nummer dieses Threads
- `void omp_set_num_threads(int i);` legt die Anzahl zu verwendender Threads fest
- `int omp_set_dynamic(int i);` Erlaube dynamische Adaption der Anzahl Threads (0 oder 1)
- `void omp_set_nested(int i);` Erlaube verschachtelten Parallelismus (0 oder 1)
- `double omp_get_wtime();` Liefert die Zeit zurück, die seit einem willkürlichen Zeitpunkt in der Vergangenheit vom Prozess verbraucht wurde. Sinnvoll sind nur die Differenzen zwischen zwei Zeitpunkten.

```
#ifdef _OPENMP
#ifdef _OPENMP
#include<omp.h>
#endif
#include<iostream>
const int CHUNK_SIZE=3;

int main(void)
{
  int id;
  std::cout << "This␣computer␣has␣" << omp_get_num_procs() << "␣
     processors" << std::endl;
  std::cout << "Allowing␣two␣threads␣per␣processor" <<
     std::endl;
  omp_set_num_threads(2*omp_get_num_procs());

  #pragma omp parallel default(shared) private(id)
  {
    #pragma omp for schedule(static,CHUNK_SIZE)
    for (int i = 0; i < 7; ++i)
    {
      id = omp_get_thread_num();
      std::cout << "Hello␣World␣from␣thread␣" << id <<
         std::endl;
    }
    #pragma omp master
      std::cout << "There␣are␣" << omp_get_num_threads() << "␣
         threads" << std::endl;
  }
  std::cout << "There␣are␣" << omp_get_num_threads() << "␣
     threads" << std::endl;

  return 0;
}
```

Output

```
This computer has 2 processors
Allowing two threads per processor
Hello World from thread 1
Hello World from thread Hello World from thread 0
Hello World from thread 0
Hello World from thread 0
2Hello World from thread
1
Hello World from thread 1
There are 4 threads
There are 1 threads
```

5.4 Locks

In `omp.h` sind auch Funktionen für die Erstellung eines Locks definiert:

- `void omp_init_lock(omp_lock_t *lock);` initialisiere lock und setze es auf unlocked.
- `void omp_destroy_lock(omp_lock_t *lock);` setze lock auf uninitialisiert.
- `void omp_set_lock(omp_lock_t *lock);` Fordere lock an und warte solange bis es verfügbar ist.
- `void omp_unset_lock(omp_lock_t *lock);` Gibt lock wieder frei.
- `int omp_test_lock(omp_lock_t *lock);` Prüfe ob lock verfügbar ist. Liefert 1 zurück, wenn das lock gesetzt werden konnte und 0 sonst.

Es gibt auch Locks die ineinander verschachtelt werden können (nested Locks).

5.5 Übersetzung und Umgebungsvariablen

OpenMP wird durch spezielle Compileroptionen aktiviert. Wenn sie nicht gesetzt sind, werden die `#pragma` Anweisungen ignoriert und ein sequentielles Programm gebaut. Für den `ICC` ist die Option `-openmp`, für den `GCC` ist sie `-fopenmp`

Die Umgebungsvariable `OMP_NUM_THREADS` legt die maximale Anzahl Thread fest. Der Aufruf der Funktion `omp_set_num_threads` im Programm hat Vorrang vor der Umgebungsvariablen.

Beispiel (mit GCC und Bash unter Linux):

```
gcc -O2 -fopenmp -o scalar_product scalarproduct.cc
export OMP_NUM_THREADS=3
./scalar_product
```

5.6 Weitere Informationen

Literatur

[1] OpenMP Spezifikation `http://openmp.org/wp/openmp-specifications/`

[2] OpenMP Tutorium/Referenz `https://computing.llnl.gov/tutorials/openMP`

[3] Intel Compiler (frei für nicht-kommerzielle Nutzung, d.h. alles wofür man nicht bezahlt wird) `https://software.intel.com/de-de/intel-education-offerings/`

6 Parallele Programmiermodelle für Shared-Memory Maschinen: Fortsetzung

6.1 Globale Synchronisation

6.1.1 Barriere

- *Barriere*: Alle Prozessoren sollen aufeinander warten bis alle angekommen sind
- Barrieren werden häufig wiederholt ausgeführt:

```
while (1) {
    eine Berechnung;
    Barriere;
}
```

- Da die Berechnung lastverteilt ist, kommen alle gleichzeitig an der Barriere an
- Erste Idee: Zähle alle ankommenden Prozesse

Programm 6.1 (Erster Vorschlag 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
        }
    }
}
```

Problem: Prozess 0 wartet auf *release*==1, Prozess 1 kommt, setzt *release*=1, läuft sofort in die nächste Barriere und setzt *release*=0 *bevor* Prozess 0 das *release*==1 gesehen hat. Folge: Verklemmung.

Barriere mit Richtungsumkehr

Warte *abwechselnd* auf *release*==1 und *release*==0

Programm 6.2 (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) ;
        }
    }
}
```

Aufwand ist $O(P^2)$ da alle P Prozesse gleichzeitig durch einen kritischen Abschnitt müssen. Geht es besser?

Hierarchische Barriere: Variante 1

Bei der Barrier mit Zähler müssen alle P Prozesse durch einen kritischen Abschnitt. Dies erfordert $O(P^2)$ Speicherzugriffe. Wir entwickeln nun eine Lösung mit $O(P \log P)$ Zugriffen.

Wir beginnen mit *zwei* Prozessen und betrachten folgendes Programmsegment:

```
int arrived=0, continue=0;

Π0:                          Π1:
                             arrived=1;
while (¬arrived) ;
arrived=0;
continue=1;
                             while (¬continue) ;
                             continue=0;
```

Wir verwenden zwei Synchronisationsvariablen, sogenannte *Flaggen*

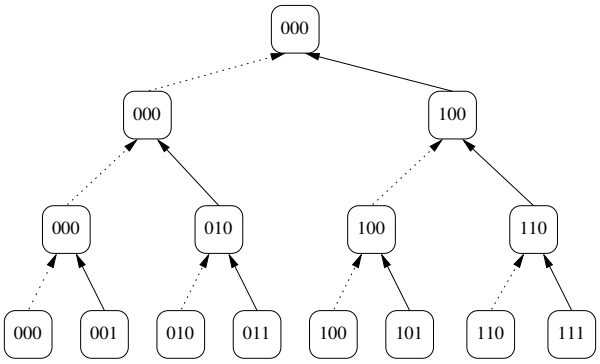
Bei Verwendung von Flaggen sind folgende Regeln zu beachten:

1. Der Prozess, der auf eine Flagge wartet setzt sie auch zurück.
2. Eine Flagge darf erst erneut gesetzt werden, wenn sie sicher zurückgesetzt worden ist.

Beide Regeln werden von unserer Lösung beachtet

Die Lösung nimmt sequentielle Konsistenz des Speichers an!

Wir wenden diese Idee nun hierarchisch an:



Programm 6.3 (Barriere mit Baum)**.**

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

process Π [***int*** $p \in \{0, ..., P-1\}$]
{
int i, r, m, k;
while *(1)* {
Berechnung;
for $(i = 0; i < d; i{+}{+})$ { // *aufwärts*
$r = p \ \& \left[\sim \left(\sum_{k=0}^{i} 2^k\right)\right]$; // *Bits 0 bis i löschen*
$m = r \mid 2^i$; // *Bit i setzen*
if $(p == m)$ *arrived[m]=1;*
if $(p == r)$ {
while$(\neg$*arrived[m]*$)$; // *warte*
arrived[m]=0;
}
} // *Prozess 0 weiss, dass alle da sind*
...
}

Programm 6.4 (Barriere mit Baum cont.)**.**
parallel *tree-barrier cont.*
{

```
    ...
        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;
        }
    }
}
```

Achtung: Flaggenvariablen sollten in verschiedenen Cache-Lines sein, damit sich Zugriffe nicht behindern!

Hierarchische Barriere: Variante 2

Diese Variante stellt eine symmetrische Lösung der Barriere mit *rekursiver Verdopplung* vor.

Wir betrachten wieder zunächst die Barriere für zwei Prozesse Π_i und Π_j:

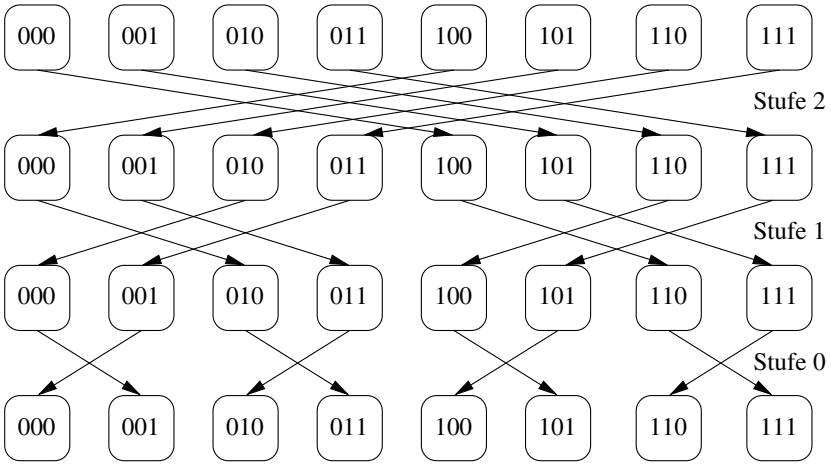
Π_i:	Π_j:
while (*arrived*[i]) ;	**while** (*arrived*[j]) ;
arrived[i]=1;	*arrived*[j]=1;
while ($\neg$*arrived*[j]) ;	**while** ($\neg$*arrived*[i]) ;
arrived[j]=0;	*arrived*[i]=0;

Im Vorgriff auf die allgemeine Lösung sind die Flaggen als Feld organisiert, zu Beginn sind alle Flaggen 0.

Ablauf in Worten:

- Zeile 2: Jeder setzt *seine* Flagge auf 1
- Zeile 3: Jeder wartet auf die Flagge des anderen
- Zeile 4: Jeder setzt die Flagge des *anderen* zurück
- Zeile 1: Wegen Regel 2 von oben warte bis Flagge zurückgesetzt ist
- Nun wenden wir die Idee rekursiv an!

Rekursive Verdopplung verwendet folgende Kommunikationsstruktur:



- Keine untätigen Prozessoren
- Jeder Schritt ist eine Zweiwegkommunikation

Programm 6.5 (Barriere mit rekursiver Verdopplung)**.**

```
parallel recursive-doubling-barrier
{
    const int d = 4,        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;
            }
        }
    }
}
```

6.1.2 Semaphore

Eine Semaphore ist eine Abstraktion einer Synchronisationsvariable, die die elegante Lösung einer Vielzahl von Synchronisationsproblemen erlaubt

Alle bisherigen Programme haben *aktives Warten* verwendet. Dies ist sehr ineffizient bei quasiparalleler Abarbeitung mehrerer Prozesse auf einem Prozessor (multitasking). Die Semaphore erlaubt es Prozesse in den Wartezustand zu versetzen.

Wir verstehen eine Semaphore als abstrakten Datentyp: Datenstruktur mit Operationen, die gewisse Eigenschaften erfüllen:

Eine Semaphore S hat einen ganzzahligen, nichtnegativen Wert $value(S)$, der beim Anlegen der Semaphore mit dem Wert *init* belegt wird.

Auf einer Semaphore S sind zwei Operationen $\mathbf{P}(S)$ und $\mathbf{V}(S)$ definiert mit:

- $\mathbf{P}(S)$ erniedrigt den Wert von S um eins falls $value(S) > 0$, sonst *blockiert* der Prozess solange bis ein anderer Prozess eine **V**–Operation auf S ausführt.
- $\mathbf{V}(S)$ befreit einen anderen Prozess aus seiner **P**–Operation falls einer wartet (warten mehrere wird einer ausgewählt), ansonsten wird der Wert von S um eins erhöht. **V**–Operationen blockieren nie!

Ist die Zahl *erfolgreich beendeter* **P**-Operation n_P und die der **V**-Operationen n_V, so gilt für den Wert der Semaphore immer:

$$value(S) = n_V + init - n_P \geq 0$$

oder äquivalent $n_P \leq n_V + init$.

Der Wert einer Semaphore ist nach aussen *nicht* sichtbar. Er äußert sich nur durch die Ausführbarkeit der **P**–Operation

Das Erhöhen bzw. Erniedrigen einer Semaphore erfolgt atomar, mehrere Prozesse können also **P**/**V**–Operationen gleichzeitig durchführen

Semaphore, die einen Wert größer als eins annehmen können bezeichnet man als *allgemeine Semaphore*

Semaphore, die nur Werte $\{0, 1\}$ annehmen, heißen *binäre Semaphore*

Notation:

Semaphore S=1;
Semaphore *forks*[5] = {1 [5]};

Wechselseitiger Ausschluss mit Semaphore

Wir zeigen nun wie alle bisher behandelten Synchronisationsprobleme mit Semaphorvariablen gelöst werden können und beginnen mit wechselseitigem Ausschluss unter Verwendung von einer einzigen binären Semaphore:

Programm 6.6 (Wechselseitiger Ausschluss mit Semaphore)**.**

```
parallel cs-semaphore
{
    const int P=8;
    Semaphore mutex=1;
    process Π [int i ∈ {0, ..., P − 1}]
    {
```

```
        while (1)
        {
            P(mutex);
            kritischer Abschnitt;
            V(mutex);
            unkritischer Abschnitt;
        }
    }
}
```

Bei Multitasking können die Prozesse in den Zustand wartend versetzt werden

Fairness ist leicht in den Aufweckmechanismus zu integrieren (FCFS)

Speicherkonsistenzmodell kann von der Implementierung beachtet werden, Programme bleiben portabel (z. B. Pthreads)

Barriere mit Semaphore

- Jeder Prozess muss verzögert werden bis der andere an der Barriere ankommt.
- Die Barriere muss wiederverwendbar sein, da sie in der Regel wiederholt ausgeführt wird.

Programm 6.7 (Barriere mit Semaphore für zwei Prozesse)**.**

```
parallel barrier-2-semaphore
{
    Semaphore b1=0, b2=0;
    process Π1                          process Π2
    {                                   {
        while (1) {                         while (1) {
            Berechnung;                         Berechnung;
            V(b1);                              V(b2);
            P(b2);                              P(b1);
        }                                   }
    }                                   }
}
```

Rollen wir die Schleifen ab, dann sieht es so aus:

```
Π1:                     Π2:
    Berechnung 1;           Berechnung 1;
    V(b1);                  V(b2);
    P(b2);                  P(b1);
    Berechnung 2;           Berechnung 2;
    V(b1);                  V(b2);
    P(b2);                  P(b1);
    Berechnung 3;           Berechnung 3;
    V(b1);                  V(b2);
    P(b2);                  P(b1);
    ...                     ...
```

Angenommen Prozess Π_1 arbeitet an Berechnungsphase i, d.h. er hat $\mathbf{P}(b2)$ $i-1$-mal ausgeführt. Angenommen Π_2 *arbeitet an* Berechnungphase $j < i$, d.h. er hat $\mathbf{V}(b2)$ $j-1$ mal ausgeführt, somit gilt

$$n_P(b2) = i - 1 > j - 1 = n_V(b2).$$

Andererseits stellen die Semaphorenregeln sicher, dass

$$n_P(b2) \leq n_V(b2) + 0.$$

Dies ist ein Widerspruch und es kann nicht $j < i$ gelten. Das Argument ist symmetrisch und gilt auch bei Vertauschen der Prozessornummern.

6.2 Beispiele

6.2.1 Erzeuger/Verbraucher

Erzeuger/Verbraucher $m/n/1$

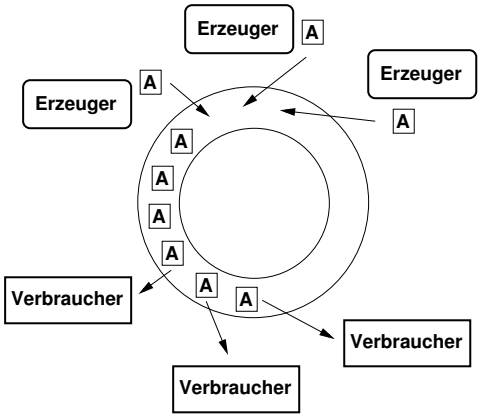
m Erzeuger, n Verbraucher, 1 Pufferplatz,

Erzeuger muss blockieren wenn Pufferplatz besetzt ist

Verbraucher muss blockieren wenn kein Auftrag da ist

Wir benutzen zwei Semaphore:

- *empty*: zählt Anzahl *freie* Pufferplätze
- *full*: zählt Anzahl *besetzte* Plätze (Aufträge)



Programm 6.8 (m Erzeuger, n Verbraucher, 1 Pufferplatz)**.**

```
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;
        }
    }
}
```

Geteilte binäre Semaphore (*split binary semaphore*):

$0 \leq empty + full \leq 1$ (Invariante)

Erzeuger/Verbraucher $1/1/k$

1 Erzeuger, 1 Verbraucher, k Pufferplätze,

Puffer ist Feld der Länge k vom Typ T. Einfügen und Löschen geht mit

$$buf[front] = t; \quad front = (front + 1) \mod k;$$

$$t = buf[rear]; \quad rear = (rear + 1) \mod k;$$

Semaphore wie oben, nur mit k initialisiert!

Programm 6.9 (1 Erzeuger, 1 Verbraucher, k Pufferplätze).

```
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

}
```

Programm 6.10 (1 Erzeuger, 1 Verbraucher, k Pufferplätze).

```
parallel prod-con-11k (cont.)
{
    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;
      }
    }
}
```

Es ist kein wechselseitiger Auschluss zur Manipulation des Puffers nötig, da P und C an verschiedenen Variablen arbeiten!

Erzeuger/Verbraucher $m/n/k$

m Erzeuger, n Verbraucher, k Pufferplätze,

Wir müssen nur sicherstellen, dass Erzeuger untereinander und Verbraucher untereinander nicht gleichzeitig den Puffer manipulieren

Benutze zwei zusätzliche binäre Semaphore *mutexP* und *mutexC*

Programm 6.11 (m Erzeuger, n Verbraucher, k Pufferplätze)**.**

```
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

}
```

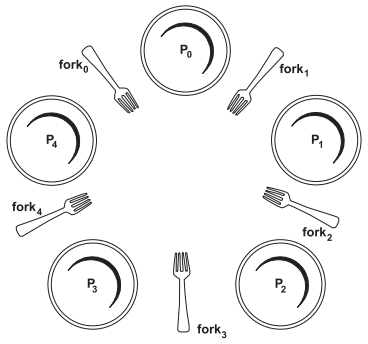
Programm 6.12 (m Erzeuger, n Verbraucher, k Pufferplätze)**.**

```
parallel 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;
        }
    }
}
```

6.2.2 Speisende Philosophen

Komplexere Synchronisationsaufgabe: Ein Prozess benötigt exklusiven Zugriff auf mehrere Resourcen um eine Aufgabe durchführen zu können. → Überlappende kritische Abschnitte.



Fünf Philosophen sitzen an einem runden Tisch. Die Tätigkeit jedes Philosophen besteht aus den sich abwechselnden Phasen des Denkens und des Essens. Zwischen je zwei Philosophen liegt eine Gabel und in der Mitte steht ein Berg Spaghetti. Zum Essen benötigt ein Philosoph zwei Gabeln – die links *und* rechts von ihm liegende.

Das Problem:
Schreibe ein paralleles Programm, mit einem Prozess pro Philosoph, welches

- einer maximalen Zahl von Philosophen zu Essen erlaubt und
- das eine Verklemmung vermeidet

Grundgerüst eines Philosophen:

```
while (1)
{
    Denke;
    Nehme Gabeln;
    Esse;
    Lege Gabeln zurück;
}
```

Naive Philosophen

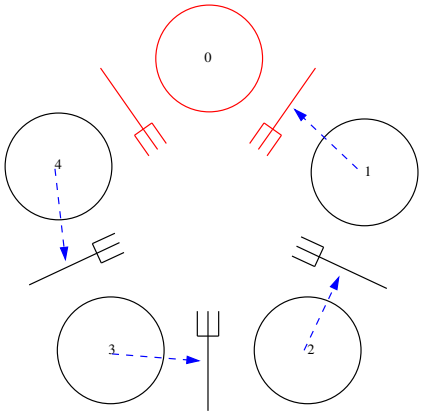
Programm 6.13 (Naive Lösung des Philosophenproblems).

```
parallel philosophers–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]);                     // linke Gabel
            P(fork[(p + 1) mod P]);         // rechte Gabel
            Esse;
            V(fork[p]);                     // linke Gabel
            V(fork[(p + 1) mod P]);         // rechte Gabel
        }
    }
}
```

Philosophen sind verklemmt, falls alle zuerst die rechte Gabel nehmen!

Einfache Lösung des Deadlockproblems: Vermeide zyklische Abhängigkeiten, z. B. dadurch, dass der Philosoph 0 seine Gabeln in der anderen Reihenfolge links/rechts nimmt.

Diese Lösung führt eventuell nicht zu maximaler Parallelität:



Schlaue Philosophen

Nehme Gabeln nur wenn *beide* frei sind

Kritischer Abschnitt: nur einer kann Gabeln manipulieren

Drei Zustände eines Philosophen: denkend, hungrig, essend

Programm 6.14 (Lösung des Philosophenproblems).

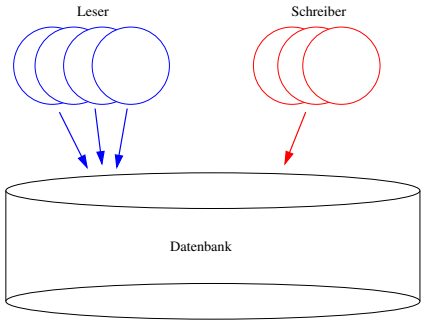
```
parallel philosophers–2
{
    const int P = 5;                    // Anzahl Philosophen
    const int think=0, hungry=1, eat=2;
    Semaphore mutex=1;
    Semaphore s[P] = { 0 [P] };         // essender Philosoph
    int state[P] = { think [P] };       // Zustand


}
```

Programm 6.15 (Lösung des Philosophenproblems).

```
parallel process
{
    Philosopher [int p ∈ {0, ..., P − 1}] {
        void test (int i) {
            int l=(i + P − 1) mod P, r=(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]);                     // warte, falls Nachbar isst
            Esse;
            P(mutex);                    // Gabeln weglegen
            state[p] = think;
            test((p + P − 1) mod P);     // wecke l. Nachbarn
            test((p + 1) mod P);         // wecke r. Nachbarn
            V(mutex);
        }
    }
}
```

6.2.3 Leser/Schreiber Problem



Zwei Klassen von Prozessen, Leser und Schreiber, greifen auf eine gemeinsame Datenbank zu. Leser führen Transaktionen aus, die die Datenbank nicht verändern. Schreiber verändern die Datenbank und benötigen exklusiven Zugriff. Falls kein Schreiber Zugriff hat können beliebig viele Leser gleichzeitig zugreifen.

Probleme:

- Verklemmungsfreie Koordination der Prozesse
- Fairness: Schließliches Eintreten der Schreiber

Naive Leser/Schreiber

Zwei Semaphore:

- *rw*: Wer hat Zugriff auf Datenbank die Leser/der Schreiber
- *mutexR*: Absicherung des Leserzählers *nr*

Programm 6.16 (Leser–Schreiber–Problem, erste 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
        }
    }

}
```

Programm 6.17 (Leser–Schreiber–Problem, erste Lösung cont.).

```
parallel process
{
    Writer [int j ∈ {0, ..., n − 1}] {
        while (1) {
            P(rw);                   // Zugriff auf DB
            schreibe Datenbank;
            V(rw);                   // gebe DB frei
        }
    }
}
```

Lösung ist nicht fair: Schreiber können verhungern

Faire Leser/Schreiber

Bearbeite wartende Prozesse nach FCFS in einer Warteschlange

Variable:

- nr, nw: Anzahl der *aktiven* Leser/Schreiber ($nw \leq 1$)
- dr, dw: Anzahl *wartender* Leser/Schreiber
- buf, $front$, $rear$: Warteschlange
- Semaphore e: Absichern des Zustandes/der Warteschlange

- Semaphore r, w: Warten der Leser/Schreiber

Programm 6.18 (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


}
```

Programm 6.19 (Leser–Schreiber–Problem, faire Lösung cont1.)**.**

```
parallel readers–writers–2 cont1.
{
    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
    }

}
```

Programm 6.20 (Leser–Schreiber–Problem, faire Lösung cont2.)**.**

```
parallel 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
        }
    }

}
```

Programm 6.21 (Leser–Schreiber–Problem, faire Lösung cont3.).

```
parallel readers–writers–2 cont3.
{
    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
        }
    }
}
```

Weiterreichen des kritischen Abschnittes (*passing the baton*):

Prozess gibt kritischen Abschnitt an genau einen anderen Prozess weiter. Nach Übergabe darf er keine weiteren Zugriffe auf Synchronisationsvariable machen.

7 Threads

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“

7.1 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

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 Ende 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

Mutex Variablen

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

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

Mutex Variable ist im Zustand frei

- Versuche in den kritischen Abschnitt einzutreten (blockierend):

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

- Verlasse kritischen Abschnitt

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

- Gebe Resourcen der Mutex Variable wieder frei

```
1 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

Bedingungsvariablen: 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

Bedingungsvariablen: 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 unbedingt erfüllt sein
- Mit einer Bedingungsvariablen sollte man nur genau ein Lock verwenden

Bedingungsvariablen: 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.

Bedingungsvariablen: Ping-Pong Beispiel

```
#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);

  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);
17  pthread_cond_destroy(&continue_cond);
    pthread_mutex_destroy(&continue_mutex);
19
    return(0);
21 }
```

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

7      pthread_mutex_lock(&arrived_mutex);
       arrived_flag = 1;
9      pthread_cond_signal(&arrived_cond);
       pthread_mutex_unlock(&arrived_mutex);
11
       pthread_mutex_lock(&continue_mutex);
13     while (continue_flag==0)
         pthread_cond_wait(&continue_cond,&continue_mutex);
15     continue_flag = 0;
       pthread_mutex_unlock(&continue_mutex);
17   }
   }
```

```
   void con (void* p) /* Consumer thread */
2  {
     int i;
4    for (i=0; i<100; i++) {
       pthread_mutex_lock(&arrived_mutex);
6      while (arrived_flag==0)
         pthread_cond_wait(&arrived_cond,&arrived_mutex);
8      arrived_flag = 0;
       pthread_mutex_unlock(&arrived_mutex);
10
       printf("pong\n");
12
       pthread_mutex_lock(&continue_mutex);
14     continue_flag = 1;
       pthread_cond_signal(&continue_cond);
16     pthread_mutex_unlock(&continue_mutex);
     }
18 }
```

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 Funktion, 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.

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

Aktive Objekte

```
class ActiveObject
{
public:
  //! constructor
  ActiveObject ();

  //! destructor waits for thread to complete
  ~ActiveObject ();

  //! action to be defined by derived class
  virtual void action () = 0;

protected:
  //! use this method as last call in constructor of derived
     class
  void start ();

  //! use this method as first call in destructor of derived
     class
  void stop ();

private:
```

```
    ...
};

#include<iostream>
#include"threadtools.hh"

Flag arrived_flag,continue_flag;

int main (int argc, char *argv[])
{
  Producer prod; // starte prod als aktives Objekt
  Consumer con;  // starte con als aktives Objekt

  return(0);
} // warte auf bis prod und con fertig sind

class Producer : public ActiveObject
{
public:
  // constructor takes any arguments the thread might need
  Producer () {
    this->start();
  }

  // execute action
  virtual void action () {
    for (int i=0; i<100; i++) {
      std::cout << "ping" << std::endl;
      arrived_flag.signal();
      continue_flag.wait();
    }
  }

  // destructor waits for end of action
  ~Producer () {
    this->stop();
  }
};

class Consumer : public ActiveObject
{
public:
  // constructor takes any arguments the thread might need
  Consumer () {
    this->start();
  }

```

```
    // execute action
10  virtual void action () {
      for (int i=0; i<100; i++) {
12      arrived_flag.wait();
        std::cout << "pong" << std::endl;
14      continue_flag.signal();
      }
16  }

18  // destructor waits for end of action
    ~Consumer () {
20    this->stop();
    }
22 };
```

Links

1 PThreads tutorial vom LLNL http://www.llnl.gov/computing/tutorials/pthreads/

2 LinuxThreads Library http://pauillac.inria.fr/~xleroy/linuxthreads/

3 Thread safety of GNU standard library http://gcc.gnu.org/onlinedocs/libstdc++/17_intro/howto.html#3

4 Resourcen zu Pthreads Funktionen http://as400bks.rochester.ibm.com/iseries/v5r1/ic2924/index.htm?info/apis/rzah

7.2 C++11-Threads

C++-11 Threads

- POSIX Threads (oder pthreads) wurden 1995 eingeführt und sind seit langem das Standardmodell für Threads auf UNIX-Computern. Sie werden über ein C Interface erzeugt und gemanagt.
- C++-11 definiert ein eigenes Threading-Konzept, das im Grunde ein Wrapper für POSIX-Threads ist. Es erlaubt jedoch in vielen Fällen multi-threadding Programme viel einfacher zu programmieren. Dabei werden die folgenden Konzepte unterstützt:

  Threads Im Header `threads` werden Thread-Klassen und Funktionen für das wiederzusammenführen von Threads definiert. Außerdem Funktionen um eine Thread-ID zu erhalten und für das die Loslösung von Threads, die dann als eigenes Programm unabhängig weiter laufen können.

  Mutual Exclusion Im Header `mutex` werden Klassen für mutexes und locks definiert (auch rekursive Varianten und solche mit Timeouts). Außerdem Funktionen die prüfen ob ein lock verfügbar ist.

Condition Variables Im Header `condition_variable` wird eine Klasse definiert, die die Koordination von mehreren Threads mit Condition Variables erlaubt.

Futures Der Header `futures` definiert Klassen und Funktionen deren Operationen nicht sofort ausgeführt werden müssen sondern asynchron zum Hauptteil des Programms ausgeführt werden können. Dies erfolgt wenn möglich parallel, wenn nicht sequentiell an der Stelle an der das Ergebnis gebraucht wird.

Atomare Operationen Die Klassen aus dem Header `atomic` erlauben es bestimmte Operationen mit Integer oder Bool Variablen sowie Pointern in einer atomaren Operation durchzuführen.

7.2.1 C++-11 Thread Erzeugung

```
#include <array>
#include <iostream>
#include <thread>

void Hello(size_t number)
{
  std::cout << "Hello␣from␣thread␣" << number << std::endl;
}

int main()
{
  const int numThreads = 5;
  std::array<std::thread,numThreads> threads;

  // starte threads
  for (size_t i = 0; i < threads.size(); ++i)
    threads[i] = std::thread(Hello, i);

  std::cout << "Hello␣from␣main\n";

  // Wiedervereinigung mit den Threads, implizite Barriere
  for (size_t i = 0; i < threads.size(); ++i)
    threads[i].join();

  return 0;
}
```

Ausgabe

```
Hello from thread Hello from thread Hello from thread Hello from thread Hello from main
0Hello from thread 213
4
```

7.2.2 Beispiel: Berechnung der Vektornorm

```
#include <array>
#include <vector>
#include <iostream>
#include <thread>

void Norm(const std::vector<double> &x, double &norm, const size_t i,
    const size_t p)
{
  size_t numElem = x.size()/p;
  size_t first = numElem * i + std::min(i,x.size()%p);
  numElem += (i<(x.size()%p)?1:0);
  double localNorm = 0.0;
```

```
    for (size_t j=0;j<numElem;++j)
13    localNorm+=x[first+j]*x[first+j];

15  norm += localNorm;  // gefaehrlich...
}
17
int main()
19 {
    const size_t numThreads = 5;
21  const size_t numValues = 1000000;
    std::array<std::thread,numThreads> threads;
23  std::vector<double> x(numValues,2.0);

25  double norm = 0.0;
    // Create threads
27  for (size_t i = 0; i < threads.size(); ++i)
      threads[i] =
          std::thread(Norm,std::cref(x),std::ref(norm),i,numThreads);
29
    // Rejoin threads with main thread, barrier
31  for (size_t i = 0; i < threads.size(); ++i)
      threads[i].join();
33
    std::cout << "Norm is:" << norm << std::endl;
35  return 0;
}
```

7.2.3 Mutual Exclusion/Locks

```
#include <iostream>
2 #include <vector>
#include <thread>
4 #include <mutex>

6 std::mutex m;

8 void e(int &sh)
{
10  m.lock();
    sh+=1; // veraendere gemeinsame Daten
12  m.unlock(); // darf nie vergessen werden
}
```

- In C++11 gibt es dafür das Konstrukt des `mutex`.
- Ein `mutex` kann einen Bereich sperren, so dass er nur von einem Prozess betreten werden kann.
- Alle anderen Prozesse müssen dann vor dem kritischen Abschnitt warten. Dieser sollte deshalb so klein wie möglich sein.
- Der gleiche Mutex muss in allen Threads verfügbar sein.

Lock Guard

```
void f(int &sh)
2 {
```

```
  std::lock_guard<std::mutex> locked{m};
4 sh+=1; // veraendere gemeinsame Daten
}
```

- Damit nicht vergessen wird ein `lock` auch wieder freizugeben (auch wenn z.B. eine Exception geworfen wird) gibt es Hilfsklassen.
- Die einfachste ist ein `lock_guard`.
- Dieser bekommt bei der Initialisierung einen `mutex`, sperrt diesen und gibt ihn wieder frei, wenn der Destruktor des `lock_guard` aufgerufen wird.
- Ein `lock_guard` sollte deshalb entweder gegen Ende einer Funktion oder in einem eigenen Block definiert werden.

Unique Lock

Ein `unique_lock` verfügt über mehr Funktionalitäten als ein `lock_guard`. Es hat z.B. Funktionen um einen `mutex` zu sperren und wieder freizugeben oder um zu testen ob ein kritischer Abschnitt betreten werden kann, so dass andernfalls etwas anderes getan werden kann. Ein `unique_lock` kann auch einen bereits gesperrten `mutex` übernehmen oder das sperren erst einmal aufschieben.

```
void g(int &sh)
2 {
  std::unique_lock<std::mutex> locked{m,std::defer_lock};  // verwalte mutex, aber
      sperre ihn nicht
4 bool successful=false;
  while (!successful)
6 {
    if (locked.try_lock()) // sperre mutex wenn moeglich
8   {
      sh+=1; // veraendere gemeinsame Daten
10    successful=true;
      locked.unlock();     // in diesem Beispiel eigentlich nicht noetig
12  }
    else
14  {
      // mache etwas anderes
16  }
  }
18 }

int main()
2 {
  const size_t numThreads=std::thread::hardware_concurrency();
4 std::vector<std::thread> threads{numThreads};

6 int result=0;
  // starte Threads
8 for (size_t i = 0; i < threads.size(); ++i)
    threads[i] = std::thread{e, std::ref(result)};
10
  // Wiedervereinigung mit dem Threads, implizite Barriere
12 for (size_t i = 0; i < threads.size(); ++i)
    threads[i].join();
14
  std::cout << "Ihre Hardware unterstuetzt " << result << " Threads" << std::endl;
16
  return 0;
18 }
```

7.2.4 Berechnung der Vektornorm mit einem Mutex

```
#include<array>
#include<vector>
#include<iostream>
#include<thread>
#include<mutex>
#include<cmath>

static std::mutex nLock;

void Norm(const std::vector<double> &x, double &norm, const size_t i,
    const size_t p)
{
  size_t numElem = x.size()/p;
  size_t first = numElem * i + std::min(i,x.size()%p);
  numElem += (i<(x.size()%p)?1:0);
  double localNorm = 0.0;
  for (size_t j=0;j<numElem;++j)
    localNorm+=x[first+j]*x[first+j];

  std::lock_guard<std::mutex>
     block_threads_until_finish_this_job(nLock);
  norm += localNorm;
}

int main()
{
  const size_t numThreads = 5;
  const size_t numValues = 10000000;
  std::array<std::thread,numThreads> threads;
  std::vector<double> x(numValues,2.0);

  double norm = 0.0;
  // starte Threads
  for (size_t i = 0; i < threads.size(); ++i)
    threads[i] =
       std::thread(Norm,std::cref(x),std::ref(norm),i,numThreads);

  // Wiedervereinigung mit den Threads
  for (size_t i = 0; i < threads.size(); ++i)
    threads[i].join();

  std::cout << "Norm is: " << sqrt(norm) << std::endl;
  return 0;
}
```

7.2.5 Berechnung der Vektornorm mit Tree Combine

Parallelisierung der Summe

- Die Berechnung der globalen Summe der Norm ist bei Verwendung eines `mutex` für die Berechnung von $s = s + t$ nicht parallel.

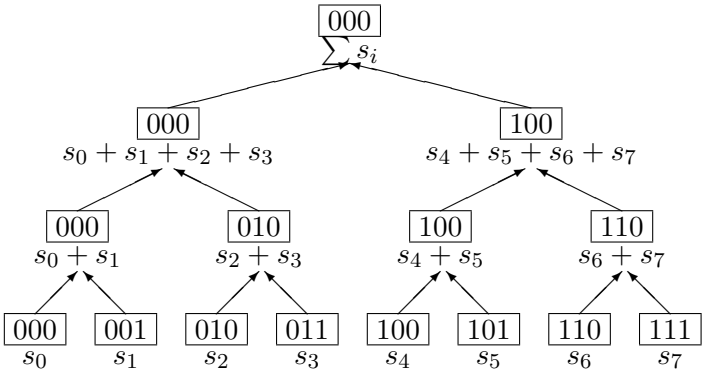
- Sie lässt sich wie folgt parallelisieren ($P = 8$):

$$s = \underbrace{\underbrace{\underbrace{s_0 + s_1}_{s_{01}} + \underbrace{s_2 + s_3}_{s_{23}}}_{s_{0123}} + \underbrace{\underbrace{s_4 + s_5}_{s_{45}} + \underbrace{s_6 + s_7}_{s_{67}}}_{s_{4567}}}_{s}$$

- Damit reduziert sich die Komplexität von $O(P)$ auf $O(\log_2 P)$.

Tree Combine

Wenn wir eine Binärdarstellung der Prozessnummer verwenden, ergibt die Kommunikationsstruktur einen binären Baum:




```
#include <array>
#include <vector>
#include <iostream>
#include <thread>
#include <cmath>

void Norm(const std::vector<double> &x, std::vector<double> &norm,
    std::vector<bool> &flag,const size_t i, const size_t d)
{
  size_t p = pow(2,d);
  size_t numElem = x.size()/p;
  size_t first = numElem * i + std::min(i,x.size()%p);
  numElem += (i<(x.size()%p)?1:0);
  for (size_t j=0;j<numElem;++j)
    norm[i]+=x[first+j]*x[first+j];

  // tree combine
  for (size_t j=0;j<d;++j)
  {
    size_t m = pow(2,j);
    if (i&m)
    {
      flag[i]=true;
      break;
    }
    while (!flag[i|m]);
```

```
26      norm[i] += norm[i|m];
    }
28  }

30  int main()
    {
32    const size_t logThreads = 1;
      const size_t numThreads = pow(2,logThreads);
34    const size_t numValues = 10000000;
      std::array<std::thread,numThreads> threads;
36    std::vector<double> x(numValues,2.0);
      std::vector<bool> flag(numThreads,false);
38
      std::vector<double> norm(numThreads,0.0);
40    // starte Threads
      for (size_t i = 0; i < threads.size(); ++i)
42      threads[i] = std::thread(Norm,std::cref(x),std::ref(norm),
                     std::ref(flag),i,logThreads);
44
      // Wiedervereinigung mit den Threads
46    for (size_t i = 0; i < threads.size(); ++i)
        threads[i].join();
48
      std::cout << "Norm␣is:␣" << sqrt(norm[0]) << std::endl;
50    return 0;
    }
```

Condition Synchronisation

- Tree combine ist eine Form der Condition Synchronisation
- Ein Prozess wartet bis eine Bedingung (boolescher Ausdruck) wahr wird. Die Bedingung wird durch einen anderen Prozess wahr gemacht.
- Hier warten mehrere Prozesse bis ihre Flags wahr werden.
- Die richtige Initialisierung der Flags ist wichtig.
- Wir implementieren die Synchronisierung mit *busy wait*.
- Das ist wahrscheinlich keine besonders gute Idee bei multi-threading.
- Die Flags werden auch *condition variables* genannt.
- Wenn condition variables wiederholt verwendet werden (z.B. wenn mehrere Summen nacheinander berechnet werden müssen) sollten die folgenden Regeln befolgt werden:
  - Ein Prozess der auf eine condition variable wartet, setzt sie auch wieder zurück.
  - Eine condition variable darf nur dann erneut wieder auf true gesetzt werden, wenn es sichergestellt ist, dass sie vorher zurückgesetzt wurde.

7.2.6 Condition Variables

Erzeuger-Verbraucher mit Condition Variable

Mit Hilfe von einer `condition_variable` und einem `unique_lock` lassen sich Threads schlafen legen und wieder aufwecken. Wird am condition variable Objekt wait aufgerufen, wartet der Thread bis er aufgeweckt wird oder er zufällig aufwacht. Eine optionale Bedingung kann ihn dann wieder schlafen legen, so lange sie nicht erfüllt ist. Es gibt auch Varianten mit Timeout.

```
#include <iostream>
#include <string>
#include <thread>
#include <mutex>
#include <condition_variable>

std::mutex m;
std::condition_variable cv;
std::string data;
bool ready = false;
bool processed = false;

void worker_thread()
{
    // Wait until main() sends data
    std::unique_lock<std::mutex> lk(m);
    cv.wait(lk, []{return ready;});

    // after the wait, we own the lock.
    std::cout << "Worker␣thread␣is␣processing␣data\n";
    data += "␣after␣processing";

    // Send data back to main()
    processed = true;
    std::cout << "Worker␣thread␣signals␣data␣processing␣
       completed\n";

    // Manual unlocking is done before notifying, to avoid
       waking up
    // the waiting thread only to block again (see notify_one
       for details)
    lk.unlock();
    cv.notify_one();
}

int main()
{
    std::thread worker(worker_thread);

    data = "Example␣data";
```

```
    // send data to the worker thread
    {
        std::lock_guard<std::mutex> lk(m);
        ready = true;
        std::cout << "main() signals data ready for
            processing\n";
    }
    cv.notify_one();

    // wait for the worker
    {
        std::unique_lock<std::mutex> lk(m);
        cv.wait(lk, []{return processed;});
    }
    std::cout << "Back in main(), data = " << data << '\n';

    worker.join();
}
```

7.2.7 Threaderzeugung mit `async`

Mit Hilfe der Funktion `async` lässt sich einfach ein Thread erzeugen, dessen Ergebnis später mit Hilfe der Funktion `get` abfragen lässt.

```
#include<iostream>
#include<vector>
#include<numeric>
#include<thread>
#include<future>

template<class T> struct Accum  {  // simple accumulator
    function object
  T *b;
  T *e;
  T val;
  Accum(T *bb, T *ee, T vv) : b{bb}, e{ee}, val{vv} {}
  T operator() () { return std::accumulate(b,e,val); }
};
```

Mit Hilfe der Funktion `async` lässt sich einfach ein Thread erzeugen, dessen Ergebnis später mit Hilfe der Funktion `get` abfragen lässt.

```
double comp(std::vector<double> &v)
  // spawn many tasks if v is large enough
{
  if (v.size()<10000)
    return std::accumulate(v.begin(),v.end(),0.0);
  std::future<double> f0
     {std::async(Accum<double>{&v[0],&v[v.size()/4],0.0})};
```

```
  std::future<double> f1
     {std::async(Accum<double>{&v[v.size()/4],&v[v.size()/2],0.0})};
  std::future<double> f2
     {std::async(Accum<double>{&v[v.size()/2],&v[v.size()*3/4],0.0})};
  std::future<double> f3
     {std::async(Accum<double>{&v[v.size()*3/4],&v[v.size()],0.0})};

  // hier koennte noch ganz viel anderer Code kommen...
  return f0.get()+f1.get()+f2.get()+f3.get();
}

int main()
{
  std::vector<double> blub(100000,1.);
  double result=comp(blub);
  std::cout << result << std::endl;
}
```

7.2.8 Threading-Unterstützung durch GCC

- Threads sind für den GCC verfügbar ab Version
  - >= 4.4.5 auf Linux-Systemen
  - >= 4.7 unter MacOS.
- Atomare Operationen sind für den GCC verfügbar ab Version
  - >= 4.5 auf Linux-Systemen
  - >= 4.7 unter MacOS.

```
g++-4.6 -std=c++0x -pthread example1.cc # syntax up to g++-4.6
g++-4.7 -std=c++11 example2.cc          # syntax from g++-4.7
clang++ -std=c++11 -stdlib=libc++ example3.cc # clang syntax
```

7.2.9 Anwendungsbeispiel: FFT

Diskrete Fourier Transformation

- Nehme Zahlenpaare (z_i, y_i) wobei y_i ein Messwert an der Stelle z_i ist.
- Berechne Interpolationspolynom

$$p(z) = \sum_{i=0}^{N} c_k e^{ikz}$$

- Dabei werden die N-ten Einheitswurzeln $z_i = e^{\frac{2\pi i}{N}k}$ als Stützstellen verwendet

- Die Koeffizienten dies Interpolationspolynoms ergeben sich als

$$c_k = \frac{1}{N} \sum_{j=0}^{N-1} y_j \cdot e^{-\frac{2\pi i}{N} jk} \quad \text{für} \quad k = 0, \ldots, N-1$$

- Dies entspricht einer Transformation des Signals in den Frequenzraum.
- Nach einer Veränderung der Frequenzanteile lässt sich das Signal wieder zurückverwandeln durch

$$y_j = \frac{1}{N} \sum_{k=0}^{N-1} c_k \cdot e^{\frac{2\pi i}{N} jk} \quad \text{für} \quad k = 0, \ldots, N-1$$

Fast Fourier Transformation

- Für ganzzahlige Werte von $N = 2n$ lässt sich die Summe

$$c_k = \sum_{j=0}^{2n-1} y_j \cdot e^{-\frac{2\pi i}{2n} jk} \quad \text{für} \quad k = 0, \ldots, 2n-1$$

umsortieren in

$$\begin{aligned} c_k &= \sum_{j=0}^{n-1} y_{2j} \cdot e^{-\frac{2\pi i}{2n} 2jk} + \sum_{j=0}^{n-1} y_{2j+1} \cdot e^{-\frac{2\pi i}{2n}(2j+1)k} \\ &= \sum_{j=0}^{n-1} y_{2j} \cdot e^{-\frac{2\pi i}{n} jk} + e^{-2\pi i \frac{k}{2n}} \sum_{j=0}^{n-1} y_{2j+1} \cdot e^{-\frac{2\pi i}{2n} 2jk} \\ &= \sum_{j=0}^{n-1} y_{2j} \cdot e^{-\frac{2\pi i}{n} jk} + e^{-\frac{\pi i}{n} k} \sum_{j=0}^{n-1} y_{2j+1} \cdot e^{-\frac{2\pi i}{n} jk} \end{aligned}$$

Fast Fourier Transformation

Es gilt also:

$$c_k^g = \sum_{j=0}^{n-1} y_{2j} \cdot e^{-\frac{2\pi i}{n} jk}, \qquad c_k^u = \sum_{j=0}^{n-1} y_{2j+1} \cdot e^{-\frac{2\pi i}{n} jk}$$

$$c_k = c_k^g + e^{-\frac{\pi i}{n} k} c_k^u$$

Wegen der $n = \frac{N}{2}$ Periodizität der Einheitswurzeln $e^{\frac{2\pi i}{n} k}$ gilt:

$$c_{k+\frac{N}{2}}^g = c_k^g \quad \text{und} \quad c_{k+\frac{N}{2}}^u = c_k^u$$

- Damit können wir das Problem jetzt durch Berechnung von zwei Fourier Transformationen der Länge $N/2$ lösen.
- Das lässt sich rekursiv anwenden. Da die Fourier-Transformierte eines Wertes der Wert selbst ist, müssen auf der untersten Stufe $N/2$ Summen aus jeweils zwei Werten berechnet werden.
- Da auf jeder Stufe N komplexe Multiplikationen mit der Einheitswurzel und N komplexe Additionen notwendig sind, lässt sich die Komplexität von $O(N^2)$ auf $O(N \cdot \log(N))$ senken.

Rekursiver Algorithmus im Pseudocode

```
procedure R_FFT(X, Y, n, w)
if (n==1) then Y[0] := X[0];
else begin
  R_FFT(<X[0],X[0],...,X[n-2]>,<Q[0],Q[1],...,Q[n/2]>,n/2,w^2);
  R_FFT(<X[1],X[3],...,X[n-1]>,<T[0],T[1],...,T[n/2]>,n/2,w^2);
  for i:= 0 to n-1 do
    Y[i] := Q[i mod (n/2)] + w^i T[i mod (n/2)];
end R_FFT
```

mit $w = e^{-\frac{2\pi i}{n}}$ aber: Rekursion schlecht parallelisierbar.

Iterativer Algorithmus im Pseudocode

```
procedure ITERATIVE_FFT(X, Y, N)
  r:= log n;
  for i := 0 to N-1 do
    R[i] := X[i];
  for i := 0 to r-1 do
    for j:= 0 to N-1 do
      S[j] := R[j];
    for j:= 0 to N-1 do
      /* Sei (b_0,b_1,...,b_r-1) die binaere Darstellung von j
         */
        k := (b_0,...,b_i-1,0,b_i+1,...,b_r-1);
        l := (b_0,...,b_i-1,1,b_i+1,...,b_r-1);
        R[j] := S[k] + S[l] * w^(b_i,b_i-1,...,b_0,0,...,0);
    endfor
  endfor
  for i := 0 to N-1 do
    Y[i] := R[i];
end ITERATIVE_FFT
```

mit $w = e^{-\frac{2\pi i}{n}}$

Sequentielle Implementierung der FFT

```
std::vector<std::complex<double>> fft(const std::vector<std::complex<double>> &data)
{
  const size_t numLevels=(size_t)std::log2(data.size());
  std::vector<std::complex<double>> result(data.begin(),data.end());
  std::vector<std::complex<double>> resultNeu(data.size());
  std::vector<std::complex<double>> root(data.size());
  for (size_t j=0;j<root.size();++j)
    root[j]=unitroot(j,root.size());
  for (size_t i=0;i<numLevels;++i)
  {
    size_t mask=1<<(numLevels-1-i);
    size_t invMask=~mask;
    for (size_t j=0;j<data.size();++j)
    {
      size_t k=j&invMask;
      size_t l=j|mask;
```

```
17        resultNeu[j]=result[k]+result[l]*root[Reversal(j/mask,i+1)*mask];
      }
19    if (i!=numLevels-1)
        result.swap(resultNeu);
21  }
  return resultNeu;
23 }
```

Zurücksortierung der Werte durch Bit-Reversal

```
1 void SortBitreversal(std::vector<std::complex<double>> &result)
  {
3   std::vector<std::complex<double>> resultNeu(result.size());
    const size_t n = std::log2(result.size());
5   for (size_t j=0;j<result.size();++j)
      resultNeu[Reversal(j,n)]=result[j];
7   result.swap(resultNeu);
  }
9
  void FastSortBitreversal(std::vector<std::complex<double>> &result)
11 {
    size_t n = result.size();
13  const size_t t = std::log2(n);
    size_t l = 1;
15  std::vector<size_t> c(n);
    for (size_t q=0;q<t;++q)
17  {
      n=n/2;
19    for(size_t j=0;j<l;++j)
        c[l+j]=c[j]+n;
21    l=2*l;
    }
23  std::vector<std::complex<double>> resultNeu(result.size());
    for (size_t j=0;j<result.size();++j)
25    resultNeu[c[j]]=result[j];
    result.swap(resultNeu);
27 }
```

Bit-Reversal und Einheitswurzeln

```
1 size_t Reversal(size_t k, size_t n)
  {
3   size_t j=0;
    size_t mask=1;
5   if (k&mask)
      ++j;
7   for (size_t i=0;i<(n-1);++i)
    {
9     mask<<=1;
      j<<=1;
11    if (k&mask)
        ++j;
13  }
    return j;
15 }

17 inline std::complex<double> unitroot(size_t i, size_t N)
  {
19  double arg = -(i*2*M_PI/N);
    return std::complex<double>(cos(arg),sin(arg));
21 }
```

Hauptprogramm Sequentielle FFT

```
int main()
{
  const size_t numPoints = pow(2,16);
  std::vector<std::complex<double>> data(numPoints);
  size_t i=1;
  for (auto &x : data)
    x.real(cos(i++));
  auto t0 = std::chrono::steady_clock::now();
  std::vector<std::complex<double>> result1 = fft(data);
  FastSortBitreversal(result1);
  auto t1 = std::chrono::steady_clock::now();
  for (auto &x : result1)
    std::cout << std::abs(x) << " " << x.real() << " " << x.imag()<< std::endl;
  std::cout << std::endl << std::endl;

  std::cout << "# Sequential fft took " <<
      std::chrono::duration_cast<std::chrono::milliseconds>(t1-t0).count() << "
      milliseconds." << std::endl;
}
```

Zeitmessung

Zur Zeitmessung müssen bisher Systembibliotheken verwendet werden. C++11 bietet eine integrierte Möglichkeit zur Zeitmessung an. Dabei gibt es verschiedene Uhren und Datentypen um Zeitpunkte und Zeitspannen zu speichern.

```
#include<chrono>
#include<iostream>
#include<thread>
using namespace std::chrono;

int main()
{
  steady_clock::time_point start = steady_clock::now();
  std::this_thread::sleep_for(seconds{2});
  auto now = steady_clock::now();
  nanoseconds duration = now-start; // we want the result in ns
  milliseconds durationMs =
     duration_cast<milliseconds>(duration);
  std::cout << "something took " << duration.count()
            << " ns which is " << durationMs.count() << " ms\n";
  seconds sec = hours{2} + minutes{35} + seconds{9};
  std::cout << "2 h 35m 9s is " << sec.count() << " s\n";
}
```

Parallele FFT: Threads

```
Barrier barrier;

void fftthread(const size_t threadNum,
    std::vector<std::complex<double>> &root,
    std::vector<std::complex<double>> &result,
    std::vector<std::complex<double>> &resultNeu)
```

```
{
  const size_t N = std::log2(result.size());
  size_t n = result.size()/numThreads;
  const size_t offset = threadNum*n +
     std::min(threadNum,result.size()%numThreads);
  n += (threadNum<(result.size()%numThreads)?1:0);
  for (size_t i=offset;i<offset+n;++i)
    root[i]=unitroot(i,root.size());
  barrier.block(numThreads);
  for (size_t level=0;level<N;++level)
  {
    size_t mask=1<<(N-level-1);
    size_t invMask=~mask;
    for (size_t i=0;i<n;++i)
    {
      size_t j=offset+i;
      size_t k=j&invMask;
      size_t l=j|mask;
      resultNeu[j]=result[k]+result[l]*root[Reversal(j/mask,level+1)*mas
    }

    barrier.block(numThreads);
    if (threadNum==0)
      result.swap(resultNeu);
    barrier.block(numThreads);
  }
  for (size_t j=offset;j<offset+n;++j)
    resultNeu[Reversal(j,N)]=result[j];
}

std::vector<std::complex<double>> fft(const
   std::vector<std::complex<double>> &data)
{
  std::vector<std::complex<double>>
     result(data.begin(),data.end());
  std::vector<std::complex<double>> resultNeu(data.size());
  std::vector<std::complex<double>> root(data.size());
  std::vector<std::thread> t(numThreads);

  for (size_t p = 0;p<numThreads;++p)
    t[p]=std::thread
       {fftthread,p,std::ref(root),std::ref(result),std::ref(resultNeu)}

  for (size_t p = 0;p<t.size();++p)
    t[p].join();

  return resultNeu;
```

```
}
```

Barrier

```
const size_t numThreads=std::thread::hardware_concurrency();

struct Barrier
{
  std::atomic<size_t> count_;
  std::atomic<size_t> step_;
  Barrier() : count_(0), step_(0)
  {}

  void block(size_t numThreads)
  {
    if (numThreads<2)
      return;
    size_t step = step_.load();
    if (count_.fetch_add(1) == (numThreads-1))
    {
      count_.store(0);
      step_.fetch_add(1);
      return;
    }
    else
    {
      while (step_.load() == step)
        std::this_thread::yield();
      return;
    }
  }
};
```

Hauptprogramm Parallele FFT

```
int main()
{
  const size_t numPoints = 2<<12;
  std::vector<std::complex<double>> data(numPoints);
  size_t i=1;
  for (auto &x : data)
    x.real(cos(i++));
  auto t0 = std::chrono::steady_clock::now();
  std::vector<std::complex<double>> result1 = fft(data);
  auto t1 = std::chrono::steady_clock::now();
  for (auto &x : result1)
    std::cout << std::abs(x) << "␣" << x.real() << "␣" <<
       x.imag()<< std::endl;
```

```
    std::cout << std::endl << std::endl;
14
    std::cout << "# Parallel fft with " << numThreads << " threads took " <<
        std::chrono::duration_cast<std::chrono::milliseconds>(t1-t0).count()
        << " milliseconds." << std::endl;
16 }
```

7.3 Weiterführende Literatur

Literatur

[1] C++11 multi-threading Tutorial http://solarianprogrammer.com/2011/12/16/cpp-11-thread-tutorial

[2] Übersicht aller C++11 Thread Klassen und Funktionen http://en.cppreference.com/w/cpp/thread

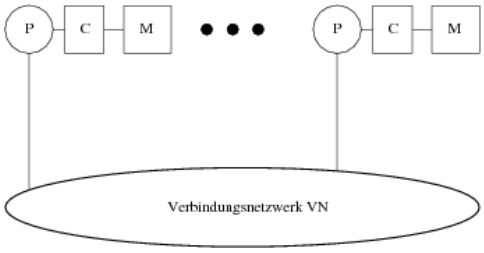
[3] Übersicht aller C++11 atomic Anweisungen http://en.cppreference.com/w/cpp/atomic

[4] Working draft des C++11 Standards (nahezu identisch mit dem endgültigen Standard aber kostenlos) http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3376.pdf

[5] Threading Building Blocks (C++ Klassenbibliothek von Intel zur vereinfachten Programmierung mit Threads) https://www.threadingbuildingblocks.org/

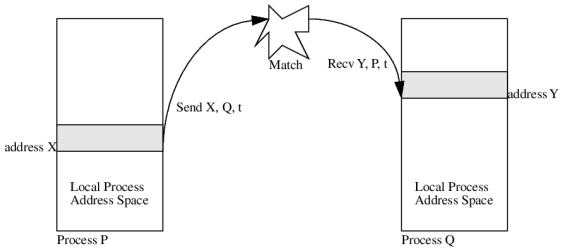
8 Computercluster und Supercomputer

Distributed Memory: Multiprozessoren



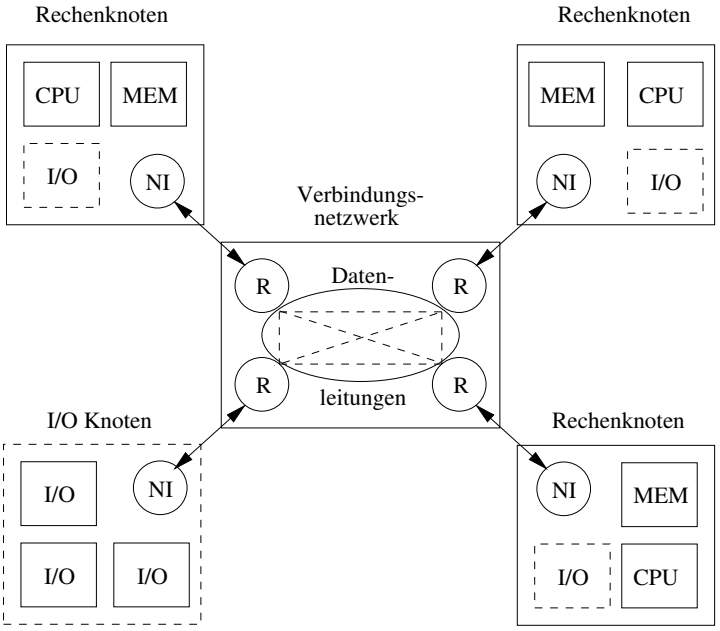
- Multiprozessoren haben einen *lokalen Adressraum*: Jeder Prozessor kann nur auf seinen Speicher zugreifen.
- Interaktion mit anderen Prozessoren ausschließlich über das Senden von Nachrichten.
- Prozessoren, Speicher und Cache sind Standardkomponenten: Volle Ausnutzung des Preisvorteils durch hohe Stückzahlen.
- Verbindungsnetzwerk von Gigabit-Ethernet bis zu proprietären High-Performance-Netzwerken.
- Ansatz mit der höchsten Skalierbarkeit: IBM BlueGene $> 10^6$ Prozesse

Distributed Memory: Message Passing



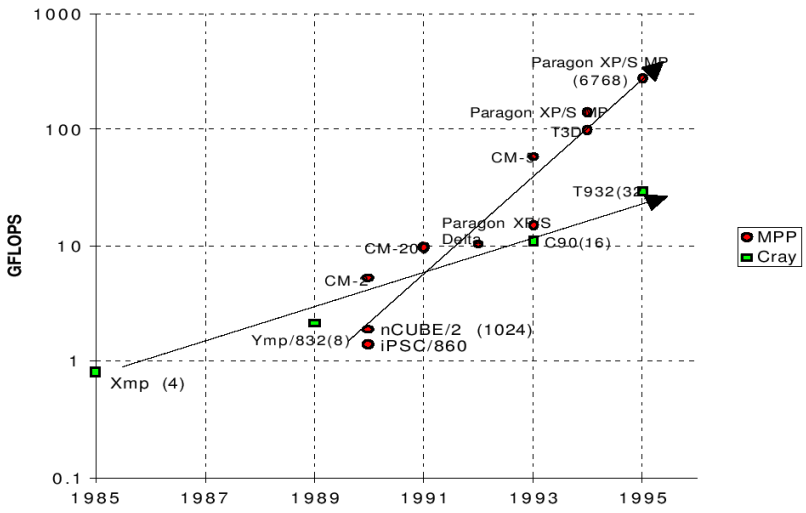
- Prozeße kommunizieren Daten zwischen verteilten Adreßräumen
- expliziter Nachrichenaustausch notwendig
- Sende-/Empfangsoperationen

Eine generische Parallelrechnerarchitektur



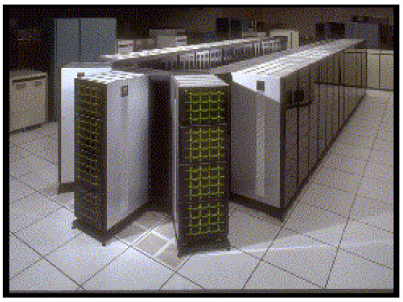
Generischer Aufbau eines skalierbaren Parallelrechners mit verteiltem Speicher

Multi-Prozessor Performance

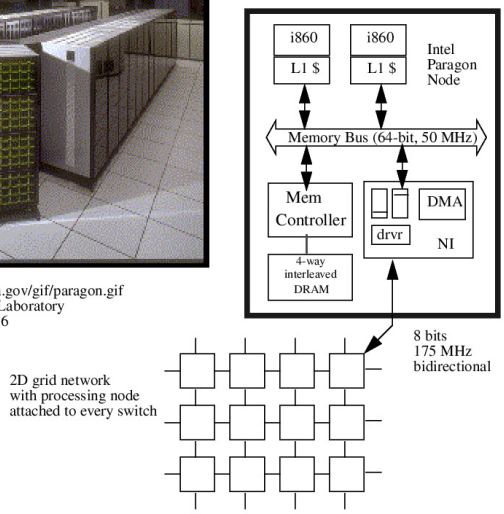


from Culler, Singh, Gupta: Parallel Computer Architecture

Fossile Rechner: Intel Paragon



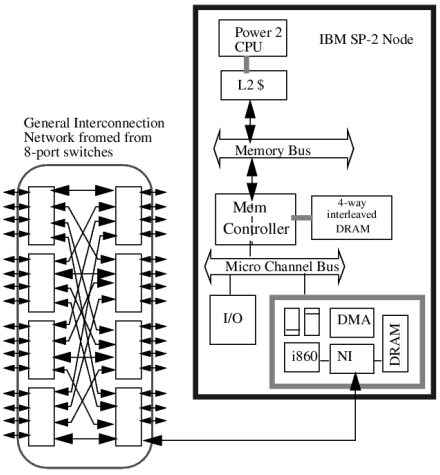
Source: http://www.cs.sandia.gov/gif/paragon.gif
courtesy of Sandia National Laboratory
1824 nodes configured as a 16
high by 114 wide array



Intel Paragon (ab 1992):

- Bis zu 4000 Rechenknoten
- Prozessmigration, Gang scheduling (gleichzeitiges Starten von Prozessen auf mehreren Maschinen)

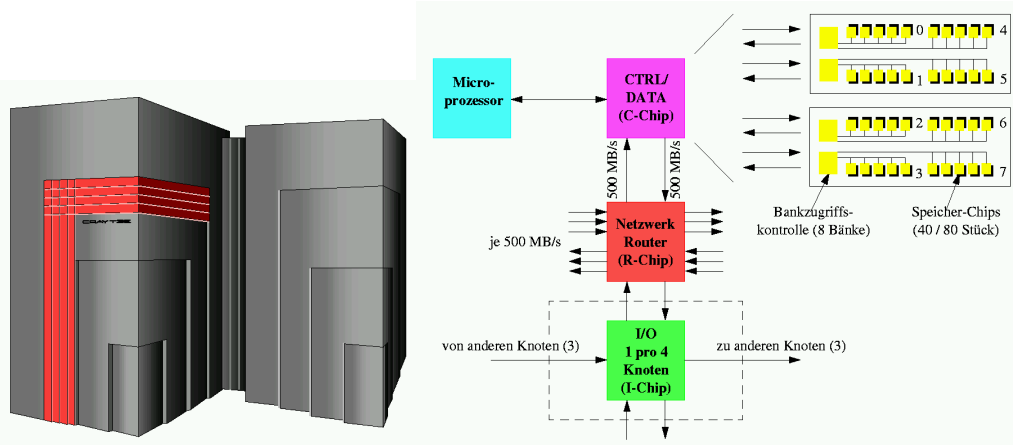
Fossile Rechner: IBM SP2



IBM SP2 (ab 1995)

- Rechenknoten: RS 6000 Workstations (bis zu 4000)
- Bis zu 8 Prozessoren pro Knoten
- am Leibniz-Rechenzentrum in München: 77 Rechenknoten mit je einer CPU
- Switching Netzwerk

Fossile Rechner Cray T3E



Cray T3E (1995)

- hohe Packungsdichte
- 3D-Torus
- virtual shared memory
- erster Terraflop-Rechner (1998)

Top500

Top500 Benchmark:

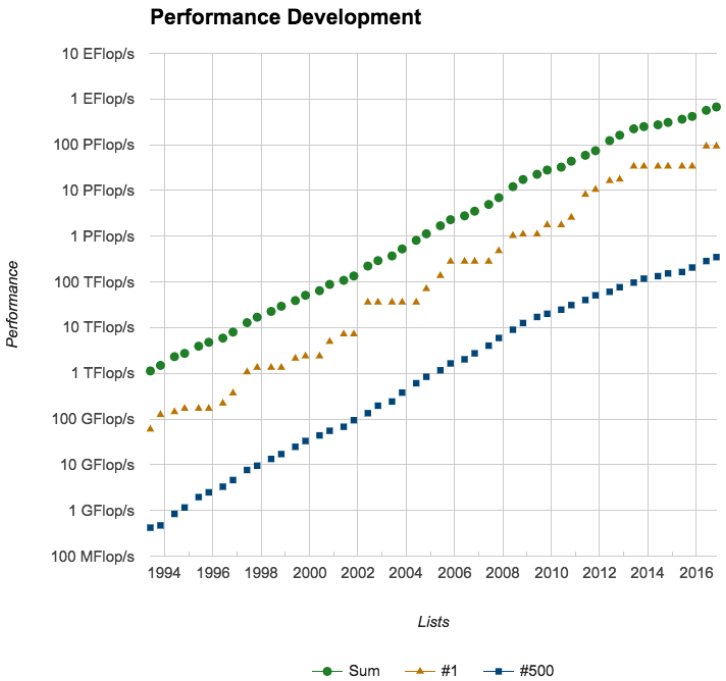
- Linpack Benchmark wird zur Evaluation der Systeme verwendet
- Benchmarkleistung reflektiert nicht die Gesamtleistung des Systems
- Benchmark zeigt Performanz bei der Lösung von dicht besetzten linearen Gleichungssystemen
- Sehr reguläres Problem: erzielte Leistung ist sehr hoch (nahe peak performance)
- Heute wird Ergänzung durch realistischeren Benchmark diskutiert: HPCG (High-Performance Conjugated Gradients), löst ein dünn-besetztes lineares Gleichungssystem.
- Schnellster Computer November 2016: Sunway TaihuLight
  - 93.0 Petaflop/s im Linpack-Benchmark
  - 0.37 Petaflop/s im HPCG-Benchmark
- Schnellster Rechner im HPCG im November 2016: Tianhe-2
  - 33.8 Petaflop/s im Linpack-Benchmark
  - 0.58 Petaflop/s im HPCG-Benchmark

Top 500 (November 2016)

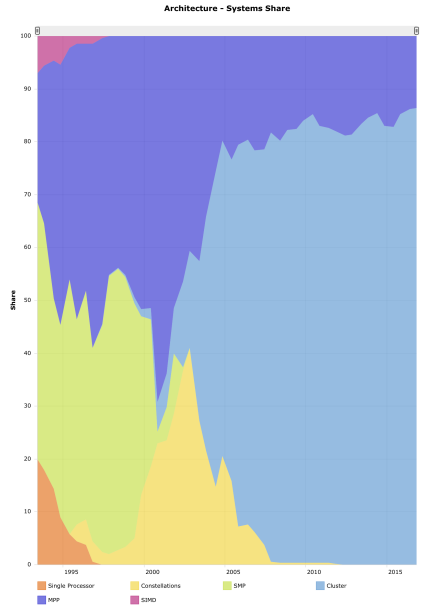
Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	National Supercomputing Center in Wuxi China	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway NRCPC	10,649,600	93,014.6	125,435.9	15,371
2	National Super Computer Center in Guangzhou China	**Tianhe-2 (MilkyWay-2)** - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT	3,120,000	33,862.7	54,902.4	17,808
3	DOE/SC/Oak Ridge National Laboratory United States	**Titan** - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.	560,640	17,590.0	27,112.5	8,209
4	DOE/NNSA/LLNL United States	**Sequoia** - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM	1,572,864	17,173.2	20,132.7	7,890
5	DOE/SC /LBNL/NERSC United States	**Cori** - Cray XC40, Intel Xeon Phi 7250 68C 1.4GHz, Aries interconnect Cray Inc.	622,336	14,014.7	27,880.7	3,939
6	Joint Center for Advanced High Performance Computing Japan	**Oakforest-PACS** - PRIMERGY CX1640 M1, Intel Xeon Phi 7250 68C 1.4GHz, Intel Omni-Path Fujitsu	556,104	13,554.6	24,913.5	2,719
7	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu	705,024	10,510.0	11,280.4	12,660

Top 500

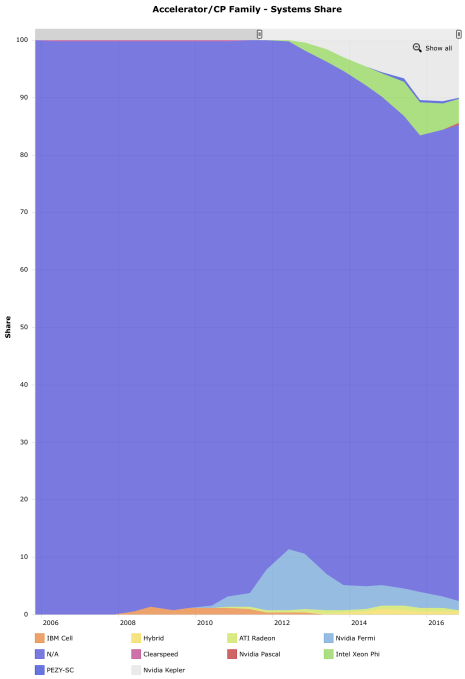
Entwicklung



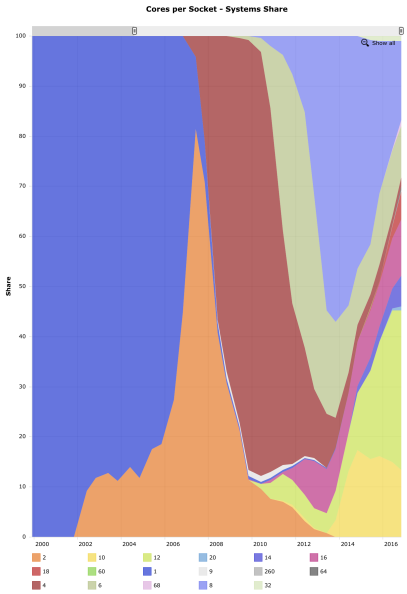
Top 500 (November 2016)
Anteil verschiedener Architekturen an den Gesamtsystemen



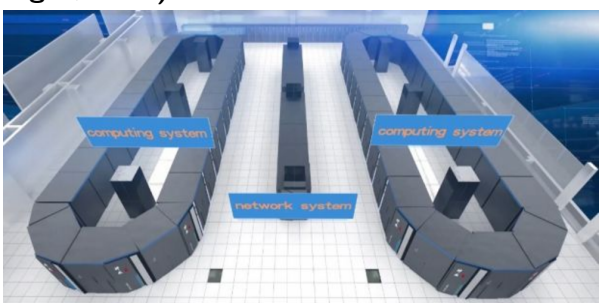
Top 500 (November 2016)
Anteil von Beschleunigerkarten an den Gesamtsystemen



Top 500 (November 2016)
Anteil Anzahl Kerne pro Prozessor an den Gesamtsystemen



Supercomputer (TaihuLight, 2016)



40'960 Knoten mit je einem Shenwei SW26010 Rechenmodul mit jeweils 256+4 64-bit RISC Kernen 1.45 GHz, 1'310'720 GB Hauptspeicher

- schnellster Rechner der Welt im Linpack (93.0 PFlop/s), drittschnellster Rechner im HPCG (0.37 PFlop/s)
- Kein Cache, 64 KB Scratchpad-Speicher für Daten, 16 KB für Instruktionen ⇒ Großes Missverhältnis zwischen Speichertransfergeschwindigkeit und Rechenleistung (22.4 Flops/-Byte Speichertransfer)
- Mellanox, Sunway Network Topologie: 256 Knoten verbunden mit 1 Superknoten, 160 Superknoten über switch verbunden
- Wasserkühlung, 15.4 MW Leisung
- Hardware komplett chinesische Eigenentwicklung

Supercomputer (Tianhe-2, 2014)

16.000 Knoten mit je 2 Intel Xeon E5-269v2 12-Core 2.2 GHz-Prozessoren, und je 3 Intel Xeon Phi 31S1P 57 Core Coprozessoren, 1'024'000 GB Hauptspeicher

- zweitschnellster Rechner der Welt im Linpack (33.9 PFlop/s), schnellster Rechner der Welt im HPCG (0.58 PFlop/s)
- proprietäres High-Performance Netzwerk, Fat-tree Topologie, 13576 Switches
- Gigabit Ethernet für Verwaltung und Überwachung
- Wasserkühlung, 17.8 MW Leisung

Supercomputer (Hazel Hen, 2015)

7'712 Knoten mit jeweils 2 Intel Xeon E5-2680v3 12-Kern CPUs (30M Cache, 2.5 GHz), 987'136 GB Hauptspeicher

- 5.6 PFlop/s (Linpack), schnellster deutscher Rechner, Platz 14 in der TOP 500 Liste (November 2016), Platz 10 im HPCG (0.14 PFlop/s)
- proprietäres High-Performance Netzwerk (Cray Aries)
- Leistung: 3.6 MW

Supercomputer (JUQUEEN, 2013)

28’672 Knoten mit IBM Power BQC 1.6 GHz 16-Core Prozessoren, 458’752 GB Hauptspeicher

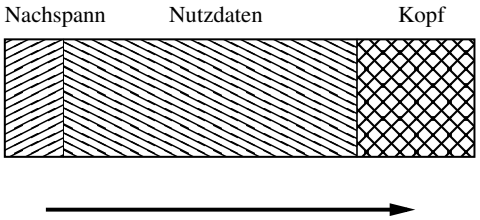
- 5.0 PFlop/s (Linpack), zweitschnellster deutscher Rechner, Platz 19 in der TOP 500 Liste, Platz 15 im HPCG (0.096 PFlop/s)
- proprietäres High-Performance Netzwerk, 5D Torus
- Gigabit Ethernet für Verwaltung und Überwachung
- 90 % Wasserkühlung, 10% Luftkühlung, 1.7 MW Leisung

9 Parallele Programmierung mit Message Passing

- Konzept wurde in den 60er Jahren entwickelt
- Ursprüngliches Ziel: Bessere Strukturierung paralleler Programme (Netzwerke gab es noch nicht)
- Idee: Daten, die ein anderer Prozess braucht werden als Nachrichten über ein Netzwerk gesendet
- Es gab früher verschiedene konkurierende Lösungen. Durchgesetzt hat sich MPI (Message Passing Interface), das seit 1994 entwickelt wird.

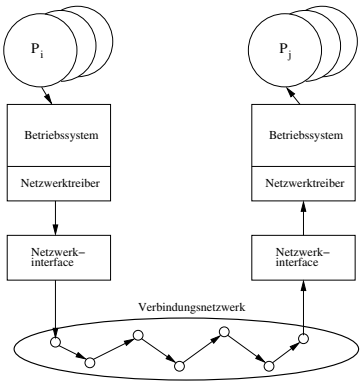
9.1 Netzwerke

- Aufgabe aus Benutzersicht: Kontinuierlicher Speicherblock (variabler Länge) soll vom Adressraum eines Prozesses in den eines anderen (i.d.R. auf einem anderen Knoten) kopiert werden.
- Das Verbindungsnetzwerk ist paketorientiert. Jede Nachricht wird in Pakete fester Länge zerlegt (z.B. 32 Byte bis 4 KByte)
- Jedes Paket besteht aus Kopf, Nachspann und Nutzdaten. Der Kopf enthält Angaben zu Zielprozessor, Größe und Art der Daten, der Nachspann z.B. eine Prüfsumme



- Das Kommunikationsprotokoll sorgt sich um die Übermittlung: Bestätigung ob Paket (richtig) angekommen ist, Flusskontrolle

Schichtenmodell (Hierarchie von Protokollen)



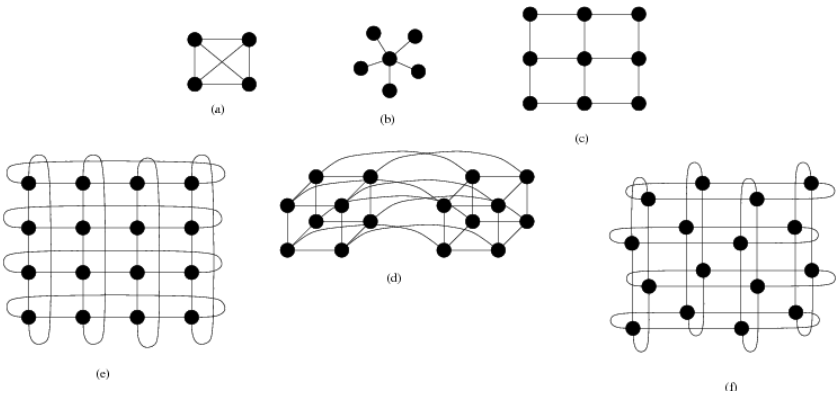
- Modell der Übertragungszeit: $t_{\mathsf{mess}}(n) = t_s + n \cdot t_b$. t_s: Setup-Zeit (Latenz), t_b: Zeit pro Byte, $1/t_b$: Bandbreite
- Die Latenz besteht insbesondere aus Software-Overhead und ist abhängig vom Kommunikationsprotokoll.
- TCP/IP ist ein störungsunanfälliges wide-area Netzwerkprotokoll. $t_s \approx 100\mu$s (unter Linux)
- Mit speziellen High-performance Netzwerken mit geringen Overhead sind Latenzen von $t_s \approx 3 \ldots 5\mu$s erreichbar

Wichtige Netzwerkeigenschaften

Netzwerkparameter, welche die Skalierbarkeit eines Systems bestimmen sind unter anderem:

- Bandbreite [MB/s], Bisektionsbandbreite
- Latenzzeit [μs]
- Robustheit gegen Ausfall einzelner Knoten oder Verbindungen (Edge-/Node-Connectivity)
- Maximale Entfernung zwischen zwei Knoten im Netzwerk (Durchmesser)
- Symmetrie des Netzwerks
- Anzahl Verbindungen pro Knoten und damit verbunden: Kosten [€]

Netzwerktopologien



(a) volle Konnektivität, (b) Star, (c) 2D-Gitter (d) 4D-Hypercube, (e) Torus, (f) Gefalteter Torus

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

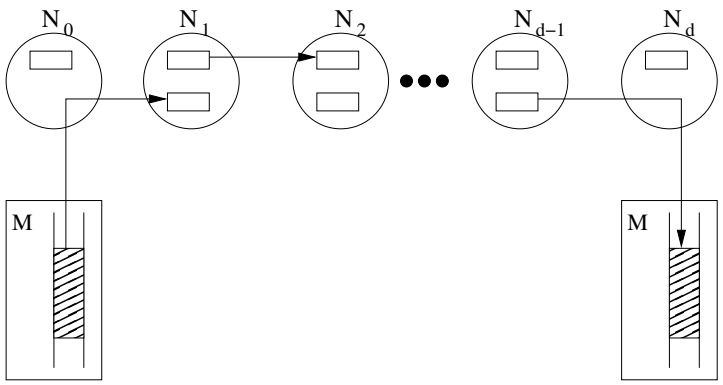
Kennzahlen (bei N Knoten):

Netzwerk-topologie	Knoten-grad K	Edge-Connectivity	Durch-messer D	Bisektions-bandbreite B	Sym-metrie
Volle Konnektivität	$N-1$	$N-1$	1	$(N/2)^2$	Ja
Stern	$N-1$	1	2	(1)	nein
Ring	2	2	$\frac{N}{2}$	2	ja
d-dim Gitter	$2d$	d	$d(\sqrt[d]{N}-1)$	$N^{\frac{d-1}{d}}$	nein
d-dim Torus	$2d$	$2d$	$d\frac{\sqrt[d]{N}}{2}$	$2N^{\frac{d-1}{d}}$	ja
d-dim. Hypercube ($d=\log_2 N$)	$\log_2 N$	$\log_2 N$	$\log_2 N$	$N/2$	ja
vollst. binärer Baum ($N=2^k-1$)	3	1	$2\log_2 \frac{N+1}{2}$	1	nein

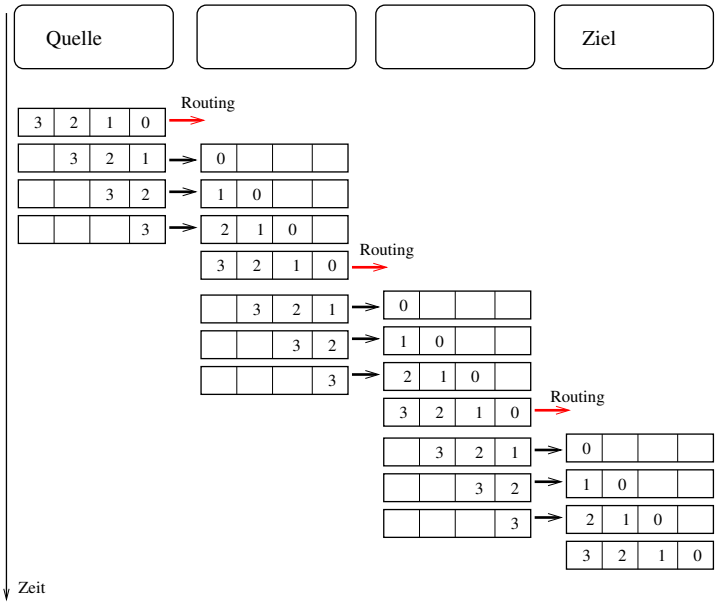
Nachrichtenübermittlung

Store & Forward Routing

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



Übertragung eines Paketes:



Laufzeit:

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

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).

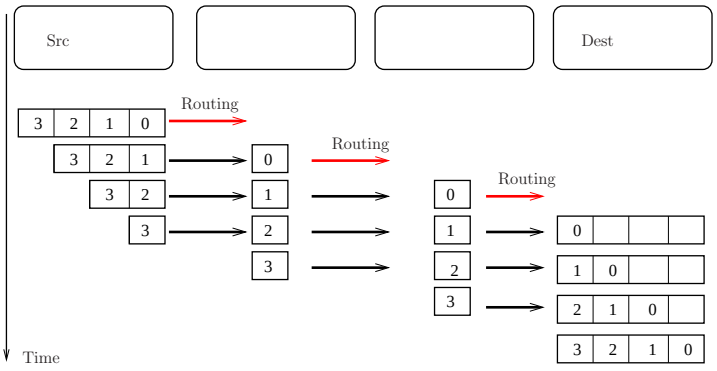
t_b: Zeit für die Übertragung eines Byte von Netzwerkknoten zu Netzwerkknoten.

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:



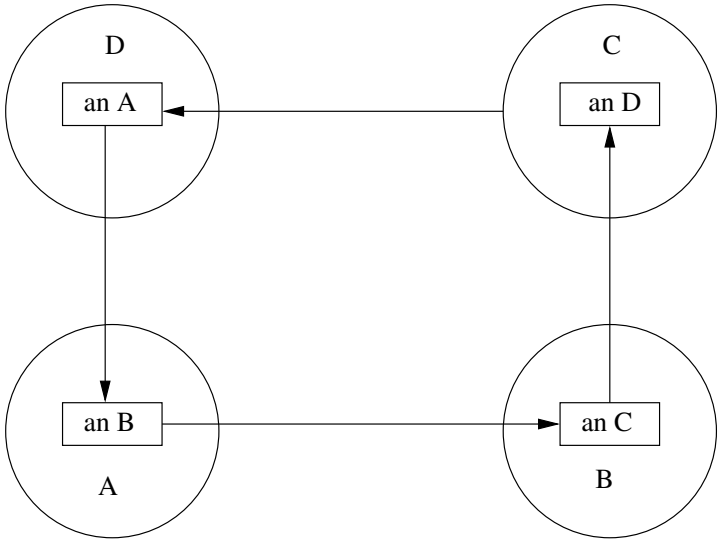
Laufzeit:

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

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

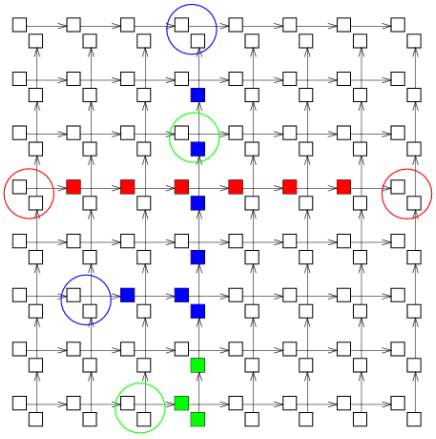
Deadlock

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



Deadlock

Zusammen mit cut-through routing:



„Dimension Routing“.

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

9.2 Nachrichtenaustausch

Um völlig auf globale Variablen verzichten zu können brauchen wir ein neues Konzept: *Nachrichten*

Syntax:

send(<Process>,<Variable>)
receive(<Process>,<Variable>)

Semantik: **send** schickt den Inhalt der Variablen an den angegebenen Prozess, **receive** wartet auf Nachricht von dem angegebenen Prozess und füllt diese in die Variable

send wartet bis die Nachricht erfolgreich empfangen wurde, **receive** blockiert den Prozess bis eine Nachricht empfangen wurde

Skalarprodukt mit Nachrichtenaustausch

Programm 9.1 (Skalarprodukt mit Nachrichtenaustausch).

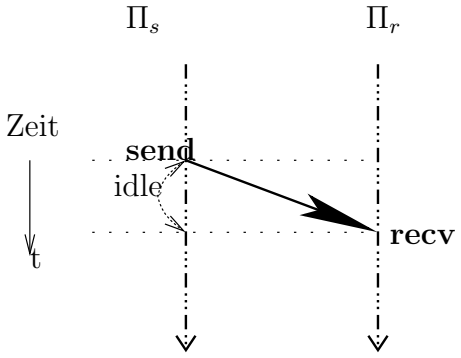
```
parallel message-passing-scalar-product
{
    const int d, P= 2^d, N;                 // Konstanten!

    process Π [int p ∈ {0,...,P − 1}]
    {
        double x[N/P], y[N/P];              // Lokaler Ausschnitt der Vektoren
        int i, r, m;
        double s, ss = 0;
        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 = log_2 P Schritte
            r = p & [∼ (∑_{k=0}^{i} 2^k)];
            m = r | 2^i;
            if (p == m)
                send(Π_r,s);
            if (p == r) {
                receive(Π_m,ls);
                s = s + ss;
            }
        }
    }
}
```

Synchronisation implizit durch **send**/**receive**

9.2.1 Synchrone Kommunikation

- Für den Nachrichtenaustausch benötigen wir mindestens zwei Funktionen:
  - **send**: Überträgt einen Speicherbereich aus dem Adressraum des Quellprozesses in das Netzwerk mit Angabe des Empfängers.
  - **recv**: Empfängt einen Speicherbereich aus dem Netzwerk und schreibt ihn in den Adressraum des Zielprozesses.

- Wir unterscheiden:
  - Zeitpunkt zu dem eine Kommunikationsfunktion beendet ist.
  - Zeitpunkt zu dem die Kommunikation wirklich stattgefunden hat.
- Bei synchroner Kommunikation sind diese Zeitpunkte identisch, d.h.
  - **send** blockiert bis der Empfänger die Nachricht angenommen hat.
  - **recv** blockiert bis die Nachricht angekommen ist.
- Syntax in unserer Programmiersprache:
  - **send**(*dest − process*,*expr*$_1$,...,*expr*$_n$)
  - **recv**(*src − process*, *var*$_1$,..., *var*$_n$)



(a)

Π_1: **recv**(Π_3,i)
send(Π_2,i)

Π_2: **recv**(Π_1,i)
send(Π_3,i)

Π_3: **recv**(Π_2,i)
send(Π_1,i)

(b)

(a) Synchronisation zweier Prozesse durch ein **send**/**recv** Paar

(b) Beispiel für eine Verklemmung

Es gibt eine Reihe von Implementierungsmöglichkeiten.

- Senderinitiiert, *three-way handshake*:
  - Quelle Q schickt *ready-to-send* an Ziel Z.
  - Ziel schickt *ready-to-receive* wenn **recv** ausgeführt wurde.
  - Quelle überträgt Nachricht (variable Länge, single copy).
- Empfängerinitiiert, *two-phase protocol*:
  - Z schickt *ready-to-receive* an Q wenn **recv** ausgeführt wurde.
  - Q überträgt Nachricht (variable Länge, single copy).
- Gepuffertes Senden
  - Q überträgt Nachricht sofort, Z muss eventuell zwischenspeichern.
  - Hier stellt sich das Problem des endlichen Speicherplatzes!

Guards

- Synchrones **send/recv** ist nicht ausreichend um alle Kommunikationsaufgaben zu lösen!
- Beispiel: Im Erzeuger-Verbraucher-Problem wird der Puffer als eigenständiger Prozess realisiert. In diesem Fall kann der Prozess nicht wissen mit welchem Erzeuger oder Verbraucher er als nächstes kommunizieren wird. Folglich kann ein blockierendes **send** zur Verklemmung führen.
- Lösung: Bereitstellung zusätzlicher *Wächterfunktionen* (Guards), die überprüfen ob ein **send** oder **recv** zur Blockade führen würde:
  - **int sprobe**($dest - process$)
  - **int rprobe**($src - process$).

  sprobe liefert 1 falls der Empfängerprozess bereit ist zu empfangen, d.h. ein **send** wird nicht blockieren:
  - **if** (**sprobe**(Π_d)) **send**(Π_d,...);

  Analog für **rprobe**.
- Wächterfunktionen blockieren nie!
- Man braucht nur eine der beiden Funktionen.
  - **rprobe** lässt sich leicht in das senderinitiierte Protokoll integrieren.
  - **sprobe** lässt sich leicht in das empfängerinitiierte Protokoll integrieren.
- Ein Befehl mit ähnlicher Wirkung wie **rprobe** ist:
  - **recv_any**(who,var_1,...,var_n).

  Er erlaubt das Empfangen von einem *beliebigen* Prozess, dessen ID in der Variablen *who* abgelegt wird.
- **recv_any** wird am einfachsten mit dem senderinitiierten Protokoll implementiert.

9.2.2 Asynchrone Kommunikation

- Befehle zum asynchronen Nachrichtenaustausch:
  - **asend**($dest - process$,$expr_1$,...,$expr_n$)
  - **arecv**($src - process$, var_1,..., var_n)
- Hier zeigt das Beenden der Kommunikationsfunktion *nicht* an, dass die Kommunikation tatsächlich stattgefunden hat. Dies muss mit extra Funktionen erfragt werden.
- Man stellt sich vor, es wird ein *Auftrag* an das System gegeben die entsprechende Kommunikation durchzuführen, sobald sie möglich ist. Der Rechenprozess kann währenddessen andere Dinge tun (*communication hiding*).
- Syntax:
  - **msgid asend**($dest - process$,var_1,...,var_n)

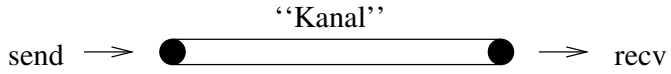
– **msgid arecv**($src - process$,var_1,...,var_n)

blockieren nie! **msgid** ist eine Quittung für den Kommunikationsauftrag.

- Vorsicht: Die Variablen var_1,...,var_n dürfen nach Absetzen des Kommunikationsbefehls nicht mehr modifiziert werden!
- Dies bedeutet, dass das Programm den Speicherplatz für die Kommunikationsvariablen selbst verwalten muss. Alternative wäre das gepufferte Senden, was aber mit Unwägbarkeiten und doppeltem Kopieraufwand verbunden ist.
- Schließlich muss man testen ob die Kommunikation stattgefunden hat (d.h. der Auftrag ist bearbeitet):
  – **int** success(**msgid** m)
- Danach dürfen die Kommunikationsvariablen modifiziert werden, die Quittung ist ungültig geworden.

Synchroner/Asynchroner Nachrichtenaustausch

- Synchrone und Asynchrone Operationen dürfen gemischt werden. Das ist auch im MPI Standard so implementiert.
- Bisherige Operationen waren *verbindungslos*.
- Alternative sind kanalorientierte Kommunikationsoperationen (oder virtuelle Kanäle):



  – Vor dem erstmaligen senden/empfangen an/von einem Prozess muss mittels **connect** eine Verbindung aufgebaut werden.
  – **send**/**recv** erhalten einen Kanal statt einen Prozess als Adresse.
  – Mehrere Prozesse können auf einen Kanal senden aber nur einer empfangen.
    * **send**($channel$,$expr_1$,...,$expr_n$)
    * **recv**($channel$,var_1,...,var_n).
- Wir werden keine kanalorientierten Funktionen verwenden.

9.3 Globale Kommunikation

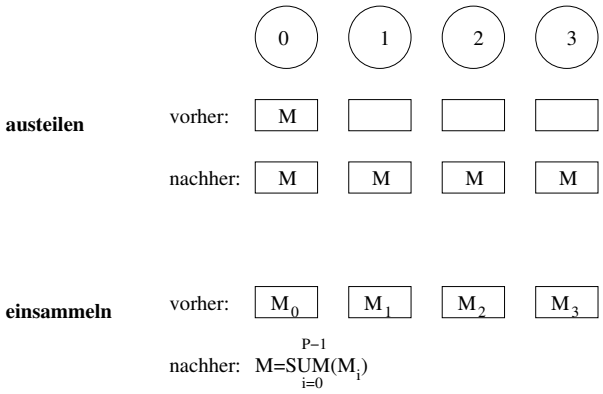
- Ein Prozess will ein *identisches* Datum an alle anderen Prozesse senden
- *one-to-all broadcast*
- duale Operation ist das Zusammenfassen von individuellen Resultaten auf einem Prozess, z.B. Summenbildung (alle assoziativen Operatoren sind möglich)
- Wir betrachten Austeilen auf verschiedenen Topologien und berechnen Zeitbedarf für store & forward und cut-through routing

- Algorithmen für das Einsammeln ergeben sich durch Umdrehen der Reihenfolge und Richtung der Kommunikationen
- Folgende Fälle werden einzeln betrachtet:
  - Einer-an-alle
  - Alle-an-alle
  - Einer-an-alle mit individuellen Nachrichten
  - Alle-an-alle mit individuellen Nachrichten

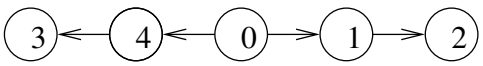
9.3.1 Einer-an-alle

Einer-an-alle: Ring

Ein Prozess will ein *identisches* Datum an alle anderen Prozesse versenden:



Hier: Kommunikation im Ring mit store & forward:



Programm 9.2 (Einer-an-alle auf dem Ring)**.**

```
parallel one-to-all-ring
{
    const int P;
    process Π[int p ∈ {0,...,P − 1}]{
        void one_to_all_broadcast(msg *mptr) {
            // Nachrichten empfangen
            if (p > 0  ∧  p ≤ P/2)          recv(Π_{p−1}, *mptr);
            if (p > P/2)                    recv(Π_{(p+1)%P}, *mptr);
            // Nachrichten an Nachfolger weitergeben
            if (p ≤ P/2 − 1)                send(Π_{p+1}, *mptr);
            if (p > P/2 + 1 ∨ p == 0)       send(Π_{(p+P−1)%P}, *mptr);
        }
```

```
        ...;
        m=...;
        one_to_all_broadcast(&m);
    }
}
```

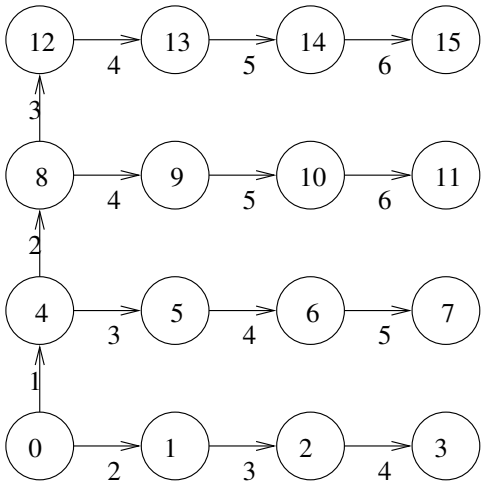
Der Zeitbedarf für die Operation beträgt (nearest-neighbour Kommunikation!):

$$T_{one-to-all-ring} = (t_s + t_h + t_w \cdot n) \left\lceil \frac{P}{2} \right\rceil,$$

wobei $n = |*mptr|$ immer die Länge der Nachricht bezeichnen soll.

Einer-an-alle: Feld

Nun setzen wir eine 2D-Feldstruktur zur Kommunikation voraus. Die Nachrichten laufen folgende Wege:



Beachte den zweidimensionalen Prozessindex.

Programm 9.3 (Einer an alle auf dem Feld).

```
parallel one-to-all-array
{
    int P, Q;                          // Feldgröße in x- und y-Richtung
    process Π[int[2] (p,q) ∈ {0,...,P−1} × {0,...,Q−1}] {
        void one_to_all_broadcast(msg *mptr) {
            if (p == 0)
            {                          // erste Spalte
                if (q > 0)             recv(Π(p,q−1),*mptr);
                if (q < Q − 1)         send(Π(p,q+1),*mptr);
            }
            else                       recv(Π(p−1,q),*mptr);
            if (p < P − 1)             send(Π(p+1,q),*mptr);
```

```
        }

        msg m=...;
        one_to_all_broadcast(&m);
    }
}
```

Die Ausführungszeit für $P = 0$ für ein 2D-Feld (Zeit pro Nachricht mal Durchmesser)

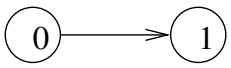
$$T_{one-to-all-array2D} = (t_s + t_h + t_w \cdot n) \cdot 2(\sqrt{P} - 1)$$

und für ein 3D-Feld

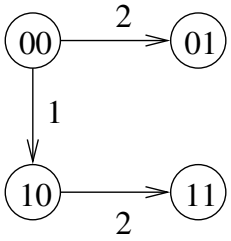
$$T_{one-to-all-array3D} = (t_s + t_h + t_w \cdot n) \cdot 3(P^{1/3} - 1).$$

Einer-an-alle: Hypercube

- Wir gehen rekursiv vor. Auf einem Hypercube der Dimension $d = 1$ ist das Problem trivial zu lösen:



- Auf einem Hypercube der Dimension $d = 2$ schickt 0 erst an 2 und das Problem ist auf 2 Hypercubes der Dimension 1 reduziert:



- Allgemein schicken im Schritt $k = 0, \ldots d-1$ die Prozesse

$$\text{je eine Nachricht an } \underbrace{p_{d-1} \ldots p_{d-k}}_{k \text{ Dimens.}} \; 0 \; \underbrace{0 \ldots 0}_{d-k-1 \text{ Dimens.}}$$
$$\underbrace{p_{d-1} \ldots p_{d-k}}_{k \text{ Dimens.}} \; 1 \; \underbrace{0 \ldots 0}_{d-k-1 \text{ Dimens.}}$$

Programm 9.4 (Einer an alle auf dem Hypercube).

```
parallel one-to-all-hypercube
{
    int d, P = 2^d;
    process Π[int p ∈ {0,...,P-1}]{
```

```
    void one_to_all_broadcast(msg *mptr) {
        int i, mask = 2^d − 1;
        for (i = d − 1; i ≥ 0; i − −) {
            mask = mask ⊕ 2^i;
            if (p&mask == 0){ Prozesse mit den letzten i Bits 0 kommunizieren
                if (p&2^i == 0)// Sende wenn das i-te Bit 0 ist
                    send(Π_{p⊕2^i}, *mptr);
                else
                    recv(Π_{p⊕2^i}, *mptr);
            }
        }
        msg m = „bla“;
        one_to_all_broadcast(&m);
    }
}
```

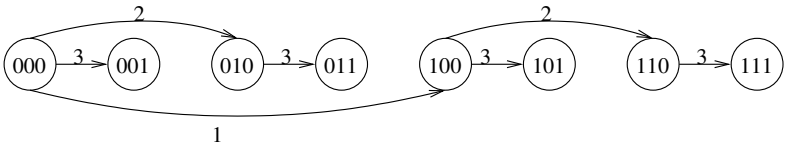
Der Zeitbedarf ist

$$T_{one-to-all-HC} = (t_s + t_h + t_w \cdot n) \operatorname{ld} P$$

Beliebige Quelle $src \in \{0, \ldots, P-1\}$: Ersetze jedes p durch $(p \oplus src)$.

Einer-an-alle: Ring mit cut-through routing

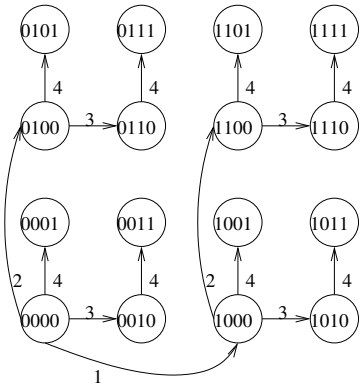
- Bildet man den Hypercubealgorithmus auf einen Ring ab, erhalten wir folgende Kommunikationsstruktur:



- Es werden keine Leitungen doppelt verwendet, somit erhält man bei cut-through routing:

$$\begin{aligned} T_{one-to-all-ring-ct} &= \sum_{i=0}^{\operatorname{ld} P - 1} (t_s + t_w \cdot n + t_h \cdot 2^i) \\ &= (t_s + t_w \cdot n) \operatorname{ld} P + t_h (P - 1) \end{aligned}$$

Bei Verwendung einer Feldstruktur erhält man folgende Kommunikationsstruktur:



Wieder gibt es keine Leitungskonflikte und wir erhalten:

$$T_{one-to-all-field-ct} = \underbrace{2}_{\substack{\text{jede} \\ \text{Entfernung} \\ \text{2 mal}}} \sum_{i=0}^{\frac{\text{ld}\,P}{2}-1} (t_s + t_w \cdot n + t_h \cdot 2^i)$$

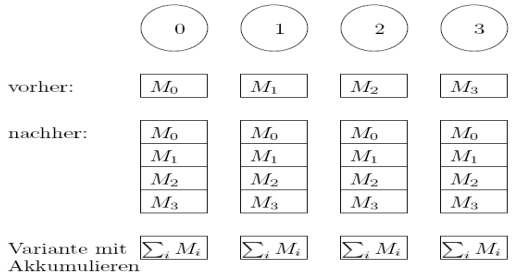
$$\begin{aligned} T_{one-to-all-field-ct} &= (t_s + t_w \cdot n) \cdot 2\frac{\text{ld}\,P}{2} + t_h \cdot 2 \underbrace{\sum_{i=0}^{\frac{\text{ld}\,P}{2}-1} 2^i}_{=\underbrace{2^{\frac{\text{ld}\,P}{2}}}_{=\sqrt{P}}-1} \\ &= \text{ld}\,P \cdot (t_s + t_w \cdot n) + t_h \cdot 2 \cdot (\sqrt{P} - 1) \end{aligned}$$

Insbesondere beim Feld ist der Term mit t_h vernachlässigbar und wir erhalten die Hypercubeperformance auch auf weniger mächtigen Topologien! Selbst für $P = 1024 = 32 \times 32$ fällt t_h nicht ins Gewicht, somit werden dank cut-through routing keine physischen Hypercube-Strukturen mehr benötigt.

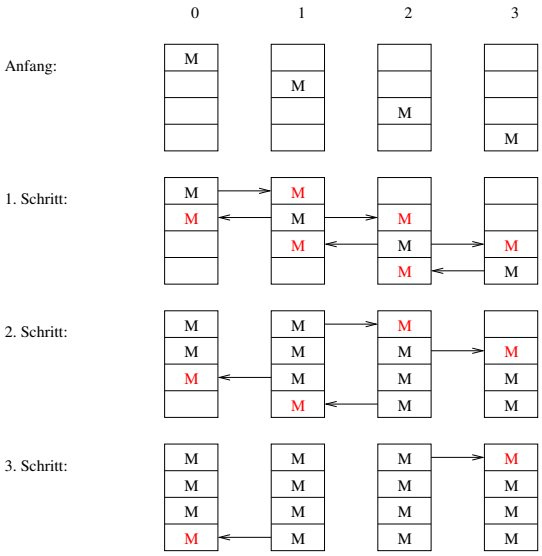
9.3.2 Alle-an-alle

Alle-an-alle: 1D Feld, Prinzip

Jeder will ein Datum an alle schicken (Variante: Akkumulieren mit assoziativem Operator):



Ring lassen wir weg und gehen gleich zum 1D-Feld: Jeder schickt in beide Richtungen.



Wir verwenden synchrone Kommunikation. Entscheide wer sendet/empfängt durch
schwarz-weiß Färbung:

Alle-an-alle: 1D Feld, Code

Programm 9.5 (Alle an alle auf 1D-Feld)**.**

```
parallel all-to-all-1D-feld
{
    const int P;
    process Π[int p ∈ {0, ..., P − 1}]
    {
        void all_to_all_broadcast(msg m[P])
        {
            int i,
                from_left= p − 1, from_right= p + 1;  // das empfange ich
                to_left= p, to_right= p;   // das verschicke ich
            for (i = 1; i < P; i++)        // P − 1 Schritte
            {
                if ((p%2) == 1)            // schwarz/weiß Färbung
                {
                    if (from_left ≥ 0) recv(Π_{p−1}, m[from_left]);
                    if (to_right ≥ 0) send(Π_{p+1}, m[to_right]);
                    if (from_right < P) recv(Π_{p+1}, m[from_right]);
                    if (to_left < P) send(Π_{p−1}, m[to_left]);
                }
                else
                {
                    if (to_right ≥ 0) send(Π_{p+1}, m[to_right]);
                    if (from_left ≥ 0) recv(Π_{p−1}, m[from_left]);
```

```
            if (to_left < P) send(Π_{p-1},m[to_left]);
            if (from_right < P) recv(Π_{p+1}, m[from_right]);
        }
        from_left--; to_right--;
        from_right++; to_left++;
      }
    }
    ...
    m[p] = „Das ist von p!“;
    all_to_all_broadcast(m);
    ...
  }
}
```

Alle-an-alle: 1D Feld, Laufzeit

- Für die Laufzeitanalyse betrachte P ungerade, $P = 2k + 1$:

$$\underbrace{\Pi_0, \ldots, \Pi_{k-1}}_{k}, \ \Pi_k, \ \underbrace{\Pi_{k+1}, \ldots, \Pi_{2k}}_{k}$$

Prozess Π_k	empfängt	k	von links
	schickt	$k+1$	nach rechts
	empfängt	k	von rechts
	schickt	$k+1$	nach links.
	$\sum =$	$4k+2$	
		$= 2P$	

- Danach hat Π_k alle Nachrichten. Jetzt muss die Nachricht von 0 noch zu $2k$ und umgedreht. Dies dauert nochmal

$$(\underbrace{k}_{\text{Entfernung}} - 1) \cdot \underbrace{2}_{\substack{\text{senden u.} \\ \text{empfangen}}} + \underbrace{1}_{\substack{\text{der Letzte} \\ \text{empfängt} \\ \text{nur}}} = 2k - 1 = P - 2$$

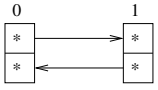
  also haben wir

$$T_{all-to-all-array-1d} = (t_s + t_h + t_w \cdot n)(3P - 2)$$

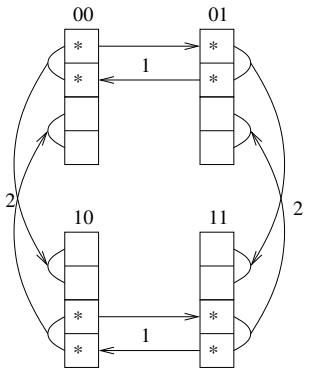
Alle-an-alle: Hypercube

Der folgende Algorithmus für den Hypercube ist als *Dimensionsaustausch* bekannt und wird wieder rekursiv hergeleitet.

Beginne mit $d = 1$:



Bei vier Prozessen tauschen erst 00 und 01 bzw 10 und 11 ihre Daten aus, dann tauschen 00 und 10 bzw 01 und 11 jeweils zwei Informationen aus



- **void** *all_to_all_broadcast*(**msg** $m[P]$) {
  int i, $mask = 2^d - 1$, q;
  for ($i = 0$; $i < d$; $i++$) {
  $q = p \oplus 2^i$;
  if ($p < q$) { // wer zuerst?
  send(Π_q,$m[p\&mask]$,...,$m[p\&mask + 2^i - 1]$);
  recv(Π_q,$m[q\&mask]$,...,$m[q\&mask + 2^i - 1]$);
  }
  else {
  recv(Π_q,$m[q\&mask]$,...,$m[q\&mask + 2^i - 1]$);
  send(Π_q,$m[p\&mask]$,...,$m[p\&mask + 2^i - 1]$);
  }
  $mask = mask \oplus 2^i$;
  }
  }

- Laufzeitanalyse:

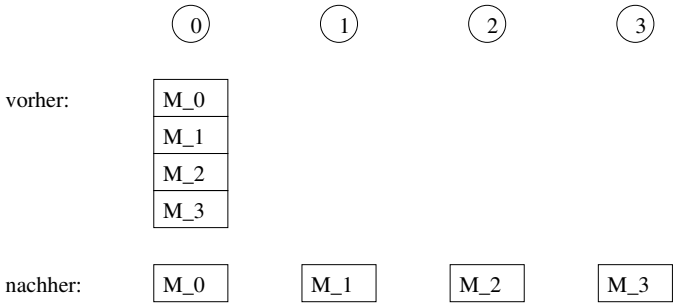
$$T_{all-to-all-bc-hc} = \underbrace{2}_{\substack{\text{send u.}\\ \text{receive}}} \sum_{i=0}^{\operatorname{ld} P-1} t_s + t_h + t_w \cdot n \cdot 2^i = \\ = 2 \operatorname{ld} P(t_s + t_h) + 2t_w n(P-1).$$

- Für große Nachrichten hat der HC keinen Vorteil gegenüber einem Torus: Jeder muss n Worte von jedem empfangen, egal wie die Topologie aussieht.

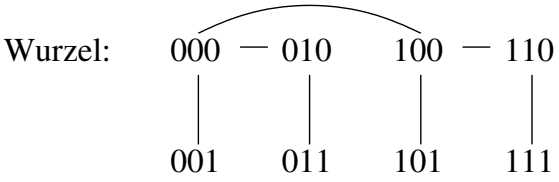
9.3.3 Einer-an-alle m. indiv. Nachrichten

Einer-an-alle indiv. Nachrichten: Hypercube, Prinzip

- Prozess 0 schickt an jeden eine Nachricht, aber an jeden eine andere!



- Beispiel ist die Ein/Ausgabe in *eine* Datei.
- Der Abwechslung halber betrachten wir hier mal die Ausgabe, d.h. alle-an-einen mit persönlichen Nachrichten.
- Wir nutzen die altbekannte Hypercubestruktur:



Einer-an-alle mit indiv. Nachrichten: Hypercube, Code

Programm 9.6 (*Einsammeln* persönlicher Nachrichten auf dem Hypercube)**.**

```
parallel all-to-one-personalized
{
    const int d, P = 2^d;
    process Π[int p ∈ {0,...,P−1}]{
        void all_to_one_pers(msg m) {
            int mask, i, q, root;
            // bestimme p's Wurzel: Wieviele Bits am Ende sind Null?
            mask = 2^d − 1;
            for (i = 0; i < d; i++)
            {
                mask = mask ⊕ 2^i;
                if (p&mask ≠ p) break;
            } // p = p_{d−1} ... p_{i+1} 1 0...0
                                  (1: zuletzt 0 gesetzt in mask; 0...0: i−1,...,0)
            if (i < d) root = p ⊕ 2^i;          // meine Wurzelrichtung

            // eigene Daten
            if (p == 0) selber-verarbeiten(m);
            else send(root,m);                  // hochgeben

            // arbeite Unterbäume ab:
            mask = 2^d − 1;
            for (i = 0; i < d; i++) {
                mask = mask ⊕ 2^i; q = p ⊕ 2^i;
                if (p&mask == p)
                                                // p = p_{d−1} ... p_{i+1} 0 0...0
                                                //                          (i−1,...,0)
                                                // q = p_{d−1} ... p_{i+1} 1 0...0
                                                //                          (i−1,...,0)
                                                // ⇒ ich bin Wurzel eines HC der Dim. i+1!
                    for (k = 0; k < 2^i; k++) {
```

```
                    recv(Π_q,m);
                    if (p == 0) verarbeite(m);
                    else send(Π_root,m);
                }
            }
        }
    }
}
```

Einer-an-alle m. indiv. Nachrichten: Laufzeit, Varianten

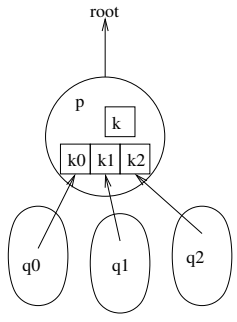
Für die *Laufzeit* hat man für grosse (n) Nachrichten

$$T_{all-to-one-pers} \geq t_w n(P-1)$$

wegen dem Pipelining.

Einige Varianten sind denkbar:

- *Individuelle Länge der Nachricht:* Hier sendet man vor verschicken der eigentlichen Nachricht nur die Längeninformation (in der Praxis ist das notwendig → MPI).
- *beliebig lange Nachricht* (aber nur endlicher Zwischenpuffer!): zerlege Nachrichten in Pakete fester Länge.
- *sortierte Ausgabe:* Es sei jeder Nachricht M_i (von Prozess i) ein Sortierschlüssel k_i zugeordnet. Die Nachrichten sollen von Prozess 0 in aufsteigender Reihenfolge der Schlüssel verarbeitet werden, *ohne* dass alle Nachrichten zwischengepuffert werden.
- Bei *sortierter Ausgabe* folgt man der folgenden Idee:

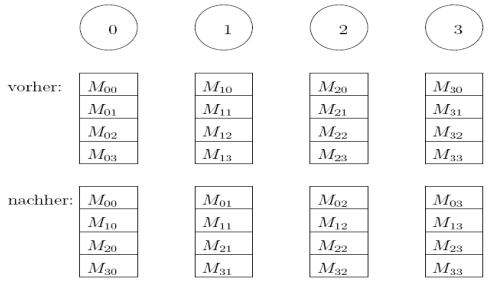


p habe drei „Untergebene“, q_0, q_1, q_2, die für ganze Unterbäume stehen.
Jeder q_i sendet seinen nächst kleinsten Schlüssel an p, der den kleinsten Schlüssel raussucht und ihn seinerseits, mitsamt der inzwischen übertragenen Daten, weitergibt.

9.3.4 Alle-an-alle m. indiv. Nachrichten

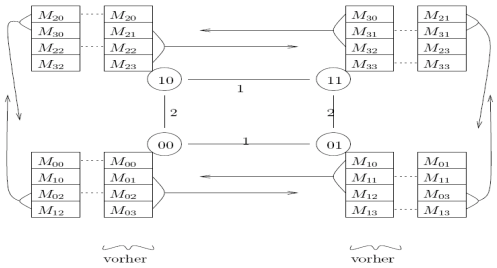
Alle-an-alle mit indiv. Nachrichten: Prinzip

Hier hat *jeder* Prozess $P-1$ Nachrichten, je eine für *jeden anderen* Prozess. Es sind also $(P-1)^2$ individuelle Nachrichten zu verschicken:



Das Bild zeigt auch schon eine Anwendung: Matrixtransposition bei spaltenweiser Aufteilung.

Wie immer, der Hypercube (hier d=2):



Alle-an-alle mit indiv. Nachrichten: Allgem. Herleitung

- Allgemein haben wir folgende Situation im Schritt $i = 0, \ldots, d-1$:
- Prozess p kommuniziert mit $q = p \oplus 2^i$ und sendet ihm

$$\begin{array}{rrl} \text{alle Daten der Prozesse } p_{d-1}\ldots p_{i+1} & p_i & x_{i-1}\ldots x_0 \\ \text{für die Prozesse } y_{d-1}\ldots y_{i+1} & \overline{p_i} & p_{i-1}\ldots p_0, \end{array}$$

  wobei die xe und $y$psilons für alle möglichen Einträge stehen.
- $\overline{p_i}$ ist Negation eines Bits.
- Es werden also in jeder Kommunikation immer $P/2$ Nachrichten gesendet.
- Prozess p besitzt zu jedem Zeitpunkt P Daten.
- Ein individuelles Datum ist von Prozess r zu Prozess s unterwegs.
- Jedes Datum ist identifiziert durch $(r, s) \in \{0, \ldots, P-1\} \times \{0, \ldots, P-1\}$.
- Wir schreiben

$$\mathcal{M}_p^i \subset \{0, \ldots, P-1\} \times \{0, \ldots, P-1\}$$

  für die Daten, die Prozess p *zu Beginn von Schritt* i besitzt, d.h. *vor* der Kommunikation.
- *Zu Beginn von Schritt 0* besitzt Prozess p die Daten

$$\mathcal{M}_p^0 = \{(p_{d-1}\ldots p_0, y_{d-1}\ldots y_0) \mid y_{d-1}, \ldots, y_0 \in \{0,1\}\}$$

- *Nach Kommunikation im Schritt* $i = 0, \ldots, d-1$ hat p die Daten $\mathcal{M}_p^{i+1}$, die sich aus $\mathcal{M}_p^i$ und folgender Regel ergeben ($q = p_{d-1}\ldots p_{i+1}\,\overline{p_i}\,p_{i-1}\ldots p_0$):

$$\begin{aligned} \mathcal{M}_p^{i+1} = \mathcal{M}_p^i & \\ \underbrace{\setminus}_{\substack{\text{schickt } p \text{ an} \\ q}} & \{(p_{d-1}\ldots p_{i+1}\,p_i\,x_{i-1}\ldots x_0, y_{d-1}\ldots y_{i+1}\,\overline{p_i}\,p_{i-1}\ldots p_0) \mid x_j, y_j \in \{0,1\}\ \forall j\} \\ \underbrace{\cup}_{\substack{\text{kriegt } p \text{ von} \\ q}} & \{(p_{d-1}\ldots p_{i+1}\,\overline{p_i}\,x_{i-1}\ldots x_0, y_{d-1}\ldots y_{i+1}\,p_i\,p_{i-1}\ldots p_0) \mid x_j, y_j \in \{0,1\}\ \forall j\} \end{aligned}$$

- *Per Induktion* gilt damit für p *nach* Kommunikation in Schritt i:

$$\mathcal{M}_p^{i+1} = \{(p_{d-1}\dots p_{i+1}x_i\dots x_0, y_{d-1}\dots y_{i+1}p_i\dots p_0) \mid x_j, y_j \in \{0,1\}\ \forall j\}$$

denn

$$\begin{aligned}
\mathcal{M}_p^{i+1} &= \big\{(p_{d-1}\dots p_{i+1} \quad p_i \quad x_{i-1}\dots x_0, \quad y_{d-1}\dots \quad y_i \quad p_{i-1}\dots p_0) \quad \mid \dots\big\} \\
&\quad \cup \big\{(p_{d-1}\dots p_{i+1} \quad \overline{p_i} \quad x_{i-1}\dots x_0, \quad y_{d-1}\dots y_{i+1} \quad p_i \quad \dots p_0) \quad \mid \dots\big\} \\
&\quad \setminus \underbrace{\{\dots\}}_{\substack{\text{was ich nicht}\\ \text{brauche}}} \\
&= \big\{(p_{d-1}\dots p_{i+1} \quad x_i \quad x_{i-1}\dots x_0, \quad y_{d-1}\dots y_{i+1} \quad p_i \quad \dots p_0) \quad \mid \dots\big\}
\end{aligned}$$

Alle-an-alle mit persönlichen Nachrichten: Code

```
void all_to_all_pers(msg m[P])
{
    int i, x, y, q, index;
    msg sbuf[P/2], rbuf[P/2];
    for (i = 0; i < d; i++)
    {
        q = p ⊕ 2^i;                    // mein Partner

        // Sendepuffer assemblieren:
        for (y = 0; y < 2^(d-i-1); y++)
            for (x = 0; x < 2^i; x++)
                sbuf[y·2^i + x] = m[y·2^(i+1) + (q&2^i) + x];
                     <P/2 (!)

        // Nachrichten austauschen:
        if (p < q)
        { send(Π_q,sbuf[0],...,sbuf[P/2-1]); recv(Π_q,rbuf[0],...,rbuf[P/2-1]); }
        else
        { recv(Π_q,rbuf[0],...,rbuf[P/2-1]); send(Π_q,sbuf[0],...,sbuf[P/2-1]); }

        // Empfangpuffer disassemblieren:
        for (y = 0; y < 2^(d-i-1); y++)
            for (x = 0; x < 2^i; x++)
                m[ y·2^(i+1) + (q&2^i) + x ] = sbuf[y·2^i + x];
                   genau das, was gesendet
                     wurde, wird ersetzt
    }
} // ende all_to_all_pers
```

Komplexitätsanalyse:

$$\begin{aligned}
T_{all-to-all-pers} &= \sum_{i=0}^{\operatorname{ld} P-1} \underbrace{2}_{\substack{\text{send u.}\\ \text{receive}}} \left(t_s + t_h + t_w \cdot \underbrace{\frac{P}{2}}_{\substack{\text{in jedem}\\ \text{Schritt}}} \cdot n\right) = \\
&= 2(t_s + t_h)\operatorname{ld} P + t_w n \cdot P \operatorname{ld} P.
\end{aligned}$$

10 MPI

Das *Message Passing Interface* (MPI) ist eine portable Bibliothek von Funktionen zum Nachrichtenaustausch zwischen Prozessen.

- MPI wurde 1993/94 von einem internationalen Gremium entworfen.
- Ist auf praktisch allen Plattformen verfügbar
- Populärste freie Implementierungen MPICH[3] und OpenMPI[4]
- Merkmale:
  - Bibliothek zum Binden mit C-, C++- und FORTRAN-Programmen (keine Sprachерweiterung).
  - Große Auswahl an Punkt-zu-Punkt Kommunikationsfunktionen.
  - Globale Kommunikation.
  - Datenkonversion für heterogene Systeme.
  - Teilmengenbildung und Topologien.
- MPI besteht aus über 125 Funktionen, die auf über 200 Seiten im Standard beschrieben werden. Daher können wir nur eine kleine Auswahl der Funktionalität betrachten.
- MPI-1 hat keine Möglichkeiten zur dynamischen Prozesserzeugung, dies ist in MPI-2 möglich, ebenso Ein-/Ausgabe.

10.1 Beispielprogrammm

Beispielprogramm in C

```
#include <stdio.h>
#include <string.h>
#include <mpi.h>  // provides MPI macros and functions

int main (int argc, char *argv[])
{
    int my_rank;
    int P;
    int dest;
    int source;
    int tag=50;
    char message[100];
    MPI_Status status;

    MPI_Init(&argc,&argv);  // begin of every MPI program

    MPI_Comm_size(MPI_COMM_WORLD,&P);  // number of
                                       // involved processes
    MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
    // number of current process always between 0 and P-1
```

[3] http://www.mpich.org
[4] http://www.open-mpi.org

```
    sprintf(message,"I␣am␣process␣%d␣of␣%d\n",my_rank,P);
    if (my_rank!=0)
    {
        dest = 0;
        MPI_Send(message,strlen(message)+1,MPI_CHAR,  // Send data
                 dest,tag,MPI_COMM_WORLD);            // (blocking)
    }
    else
    {
        puts(message);
        for (source=1; source<P; source++)
        {
            MPI_Recv(message,100,MPI_CHAR,source,tag,  // Receive data
                     MPI_COMM_WORLD,&status);          // (blocking)
            puts(message);
        }
    }

    MPI_Finalize();  // end of every MPI program

    return 0;
}
```

Übersetzen des Programms

- Das Beispielprogramm ist im SPMD-Stil geschrieben (Single Program Multiple Data). Das ist vom MPI-Standard nicht vorgeschrieben, vereinfacht aber das Starten des Programms.
- Übersetzung, Linken und Ausführung kann sich abhängig von der konkreten Implementierung unterscheiden.
- Viele Implementierungen enthalten Shellskripten, die die Lage der Bibliotheken und Header im Dateisystem verstecken und die Pfade automatisch korrekt setzen. Für MPICH muss man folgendes ausführen um das Programm zu übersetzen und 8 Prozesse zu starten:

  ```
  mpicc -o hello hello.c
  mpirun -machinefile machines -np 8 hello
  ```

  In diesem Beispiel werden die Namen der Rechner auf denen Prozesse gestartet werden der Datei`machines` entnommen.

  Für C++-Programme ist die Anweisung zur Übersetzung

  ```
  mpicxx -o hello hello.c
  ```

Output of the Example Programs (with P=8)

```
I am process 0 of 8

I am process 1 of 8

I am process 2 of 8

I am process 3 of 8
```

```
 9 I am process 4 of 8

11 I am process 5 of 8

13 I am process 6 of 8

15 I am process 7 of 8
```

Aufbau von MPI-Nachrichten

- MPI-Nachrichten bestehen aus den eigentlichen *Daten* und einer *Hülle* die folgendes enthält:
  1. Nummer des Senders,
  2. Nummer des Empfängers,
  3. Tag,
  4. und einem Kommunikator.
- Nummer des Senders und Empfängers wird als Rank bezeichnet.
- Tag ist auch eine Integer-Zahl und dient der Kennzeichnung verschiedener Nachrichten zwischen identischen Kommunikationspartnern.
- Ein Kommunikator steht für eine Teilmenge der Prozesse und einen Kommunikationskontext. Nachrichten, die zu verschiedenen Kontexten gehören, beeinflussen einander nicht, bzw. Sender und Empfänger müssen den selben Kommunikator verwenden. Der Kommunikator `MPI_COMM_WORLD` ist von MPI vordefiniert und enthält alle gestarteten Prozesse.

Initialisierung and Beenden

```
int MPI_Init(int *argc, char ***argv)
```

Bevor die ersten MPI-Funktionen verwendet werden, muss `MPI_Init` aufgerufen werden.

```
int MPI_Finalize(void)
```

Nach dem letzten MPI-Funktionsaufruf muss `MPI_Finalize` ausgeführt werden um alle Prozesse wohldefiniert herunterzufahren.

10.2 Kommunikatoren

In allen MPI Kommunikationsfunktionen tritt ein Argument vom Typ `MPI_Comm` auf. Ein solcher *Communicator* beinhaltet die folgenden Abstraktionen:

- *Prozessgruppe:* Ein Kommunikator kann benutzt werden, um eine Teilmenge aller Prozesse zu bilden. Nur diese nehmen dann etwa an einer globalen Kommunikation teil. Der vordefinierte Kommunikator `MPI_COMM_WORLD` besteht aus allen gestarteten Prozessen.

- *Kontext:* Jeder Kommunikator definiert einen eigenen Kommunikationskontext. Nachrichten können nur innerhalb des selben Kontextes empfangen werden, in dem sie abgeschickt wurden. So kann etwa eine Bibliothek numerischer Funktionen einen eigenen Kommunikator verwenden. Nachrichten der Bibliothek sind dann vollkommen von Nachrichten im Benutzerprogramm abgeschottet, und Nachrichten der Bibliothek können nicht fälschlicherweise vom Benutzerprogramm empfangen werden und umgekehrt.

- *Virtuelle Topologien:* Ein Kommunikator steht für eine Menge von Prozessen $\{0, \ldots, P-1\}$. Optional kann man diese Menge mit einer zusätzlichen Struktur, etwa einem mehrdimensionalen Feld oder einem allgemeinen Graphen, versehen.

- *Zusätzliche Attribute:* Eine Anwendung (z.B. eine Bibliothek) kann mit einem Kommunikator beliebige statische Daten assoziieren. Der Kommunikator dient dann als Vehikel, um diese Daten von einem Aufruf der Bibliothek zum nächsten hinüberzuretten.

Varianten von Kommunikatoren

Es gibt verschiedene Arten von Kommunikatoren:

`Intracomm` für die Kommunikation innerhalb einer Gruppe von Prozessen. `MPI_COMM_WORLD` ist ein Intrakommunikator da alle Prozesse Teil von `MPI_COMM_WORLD` sind. Es gibt Unterklassen von Intrakommunikatoren für die Bildung von Prozesstopologien:

`Cartcomm` kann Prozesse beinhalten, die eine kartesische Topologie haben.

`Graphcomm` kann Prozesse beinhalten, die entlang beliebiger Graphen verknüpft sind.

`Intercomm` für die Kommunikation zwischen Prozessgruppen.

Erzeugen von Kommunikatoren für Untergruppen

Ein neuer Kommunikator kann mit Hilfe der Funktion

```
int MPI_Comm_split(MPI_Comm comm, int colour,
                   int key, MPI_Comm *newcomm);
```

erzeugt werden.

`MPI_Comm_split` ist eine kollektive Operation, die von *allen* Prozessen des Kommunikators `comm` aufgerufen werden muss. Alle Prozesse mit gleichem Wert für das Argument `colour` bilden jeweils einen neuen Kommunikator. Die Reihenfolge (rank) innerhalb des neuen Kommunikator wird durch das Argument `key` geregelt.

10.3 Blockierende Kommunikation

```
int MPI_Send(void *buf, int count, MPI_Datatype dt,
             int dest, int tag, MPI_Comm comm);
int MPI_Recv(void *buf, int count, MPI_Datatype dt,
             int src, int tag, MPI_Comm comm,
             MPI_Status *status);
```

Die ersten drei Argumente `buf`, `count` und `dt` beschreiben die zu sendenden Daten. `buf` zeigt auf einen zusammenhängenden Speicherbereich, der `count` Elemente vom Datentyp `dt` enthält. Die Angabe des Datentyps ermöglicht die automatische Datenkonvertierung durch MPI.

Die Argumente `dest/src`, `tag` und `comm` bilden die Hülle der Nachricht (die Nummer des Senders/Empfängers wird beim Aufruf automatisch hinzugefügt).

Datenkonvertierung

MPI-Implementierungen für heterogene Systeme können eine automatische Konvertierung des Datenformats vornehmen. Wie das passiert bleibt der jeweiligen Implementierung vorbehalten.

MPI stellt die folgenden architekturunabhängigen Datentypen bereit:

```
MPI_CHAR, MPI_UNSIGNED_CHAR, MPI_BYTE
MPI_SHORT, MPI_INT, MPI_LONG, MPI_LONG_LONG_INT,
MPI_UNSIGNED, MPI_UNSIGNED_SHORT, MPI_UNSIGNED_LONG,
MPI_FLOAT, MPI_DOUBLE, MPI_LONG_DOUBLE
```

Der MPI Datentyp `MPI_BYTE` wird *niemals* konvertiert.

Status

```
typedef struct {
    int count;
    int MPI_SOURCE;
    int MPI_TAG;
    int MPI_ERROR;
} MPI_Status;
```

In C ist `MPI_Status` ein Struct der Information über die Anzahl empfangener Objekte, den Rank des sendenden Prozessors, das Tag und einen Fehlerindikator enthält.

Varianten von Send

- *buffered send* (`MPI_Bsend`):

  Falls der Empfänger noch keine entsprechende **recv**-Funktion ausgeführt hat, wird die Nachricht auf Senderseite gepuffert. Ein "buffered send" wird immer sofort beendent, falls genügend Pufferplatz zur Verfügung steht. Im Unterschied zur asynchronen Kommunikation kann der Sendepuffer `message` sofort wiederverwendet werden.

- *synchronous send* (`MPI_Ssend`): Die Beendigung des synchronous send zeigt an, dass der Empfänger eine **recv**-Funktion ausführt und begonnen hat, die Daten zu lesen.

- *ready send* (`MPI_Rsend`): Ein ready send darf nur ausgeführt werden, falls der Empfänger bereits das entsprechende **recv** ausgeführt hat. Ansonsten führt der Aufruf zum Fehler.

`MPI_Send` und `MPI_Receive` II

Der `MPI_Send`-Befehl hat entweder die Semantik von `MPI_Bsend` oder `MPI_Ssend`, je nach Implementierung. `MPI_Send` kann also, muss aber nicht blockieren. In jedem Fall kann der Sendepuffer `message` sofort nach beenden wieder verwendet werden.

Der Befehl `MPI_Recv` ist in jedem Fall blockierend, d.h. die Funktion wird nur beendet, wenn `buf` tatsächlich die gesendeten Daten enthält (solange kein Fehler auftritt). Das Argument `status` enthält wieder Quelle, Tag und Fehlerstatus der empfangenen Nachricht.

Für die Argumente `src` und `tag` können die Werte `MPI_ANY_SOURCE` bzw. `MPI_ANY_TAG` eingesetzt werden. Somit beinhaltet `MPI_Recv` die Funktionalität von **recv_any**.

Wächterfunktionen

```
int MPI_Iprobe(int source, int tag, MPI_Comm comm,
2                int *flag, MPI_Status *status);
```

ist eine nicht-blockierende Wächterfunktion für das Empfangen von Nachrichten. `flag` wird auf `true` ($\neq 0$ in C) gesetzt, wenn eine Nachricht mit passendem Senderrank und passendem Tag empfangen werden kann.

Auch hier sind die Argumente `MPI_ANY_SOURCE` und `MPI_ANY_TAG` wieder erlaubt.

10.4 Nichtblockierende Kommunikation

Für nichtblockierende Kommunikation stehen die Funktionen

```
int MPI_ISend(void *buf, int count, MPI_Datatype dt,
2               int dest, int tag, MPI_Comm comm,
                MPI_Request *req);
4   int MPI_IRecv(void *buf, int count, MPI_Datatype dt,
                int src, int tag, MPI_Comm comm,
6               MPI_Request *req);
```

zur Verfügung. Die Argumente entsprechen im wesentlichen denen der blockierenden Varianten.

Mittels der `MPI_Request`-Objekte ist es möglich, den Zustand des Kommunikationsauftrages zu ermitteln (entspricht unserer **msgid**).

Dazu gibt es (unter anderem) die Funktion

```
int MPI_Test(MPI_Request *req,int *flag, MPI_Status *status);
```

Das `flag` wird auf `true` ($\neq 0$) gesetzt, falls die Kommunikationsoperation abgeschlossen ist, auf die `req` verweist. In diesem Fall enthält `status` Angaben über Sender, Empfänger und Fehlerstatus.

Zu beachten ist dabei, dass das `MPI_Request`-Objekt ungültig wird, sobald `MPI\_Test` den Wert `flag==true` zurückgegeben hat. Es darf dann nicht mehr verwendet werden. Die `MPI_Request`-Objekte werden von der MPI-Implementierung selbstständig verwaltet (sogenannte opake Objekte).

10.5 Globale Kommunikation

MPI stellt auch Funktionen für Kommunikationsoperationen zur Verfügung an denen alle Prozesse eines Kommunikators teilnehmen.

```
int MPI_Barrier(MPI_Comm comm);
```

blockiert jeden einzelnen Prozess bis alle Prozesse angekommen sind (d.h. bis sie diese Funktion aufgerufen haben).

```
1   int MPI_Bcast(void *buf, int count, MPI_Datatype dt,
                int root, MPI_Comm comm);
```

verteilt die Nachricht in Prozess `root` an alle anderen Prozesse des Kommunikators.

Einsammeln von Daten

Für das Einsammeln von Daten stehen verschiedene Operationen zur Verfügung. Wir beschreiben nur eine davon:

```
int MPI_Reduce(void *sbuf, void *rbuf, int count,
2         MPI_Datatype dt, MPI_Op op, int root, MPI_Comm comm);
```

kombiniert die Daten im Eingangspuffer `sbuf` aller Prozesse mittels der assoziativen Operation `op`. Das Endergebnis steht im Empfangspuffer `rbuf` des Prozesses `root` zur Verfügung. Beispiele für `op` sind `MPI_SUM`, `MPI_PROD`, `MPI_MAX`. und `MPI_MIN` mit denen sich die Summe, das Maximum oder das Minimum eines Wertes über alle Prozesse berechnen lässt.

Anmerkung
Globale Kommunikationsfunktionen müssen mit passenden Argumenten aufgerufen werden (z.B. muss `root` in `MPI_Reduce` für alle Prozesse eines Kommunikators gleich sein).

10.6 Debugging eines Parallelen Programms

Das Debugging paralleler Programme ist keine einfache Aufgabe. Mögliche Hilfsmittel sind:

- Die Verwendung von `printf` oder `std::cout`
- Schreiben von log-files
- Verwendung von `gdb`
- Verwendung spezialisierter Debugger für parallele Programme

Verwendung von `printf` oder `std::cout`

```
std::cout << omp_get_thread_num() << ": a = " << a << std::endl;
2 // alternativ bekommt man den Rang natuerlich von MPI
```

- Der Rank (die Nummer) des Threads sollte der Ausgabe vorangestellt werden.
- Die Ausgabe verschiedener Prozesse wird gemischt und vom Root-Prozess geschrieben.
- Die Reihenfolge der Ausgabe verschiedener Prozesse muss nicht chronologisch sein.
- Es kann sein, dass am Ende des Jobs ein Teil der Ausgabe nie geschrieben wird.

Verbesserte Variante

```
#include<sstream>
2 int myRank_;
template<class T>
4 void DebugOut(const std::string message, T i) const
{
6   std::ostringstream buffer;
   buffer << myRank_ << ": " << message << " " << i << std::endl;
8   std::cout << buffer.str();
}
```

- Verwendung:

```
DebugOut("blablablub", value);
```

- Vorteil: Der Inhalt einzelner Zeilen wird nicht gemischt

Schreiben von log-files

```
#include<fstream>
#include<sstream>
int myRank_;
template<class T>
void DebugOut(const std::string message, T i) const
{
  std::ostringstream buffer;
  buffer << "debugout" << myRank_;
  std::ofstream outfile(buffer.str().c_str(),std::ios::app);
  outfile << message << "␣" << i<< std::endl;
}
```

- Verwendung:

```
DebugOut("blablablub", value);
```

- Ausgabe jedes Prozesses landet in separater Datei.
- Die Ausgabe ist für jeden Prozess vollständig (da die Datei immer unmittelbar nach dem Schreiben geschlossen wird).
- Die Ausgaben werden immer an die Datei angehängt, die Datei wird nicht gelöscht ⇒ In der Datei kann die Ausgabe von mehreren Programmläufen stehen.

Verwendung von gdb (mit Message Passing)

Mehrere Instanzen von `gdb` können gestartet werden indem man sie an einen parallelen Prozesse andocked (attached).

```
gdb <program name> <PID>
```

PID ist die Prozess-ID, die man mit dem Kommando `ps` herausfinden kann.

Um sicherzustellen, dass alle Prozese am Programmanfang anhalten, kann man eine Endlosschleife an den Anfang setzen:

```
bool debugStop=true;
while (debugStop);
```

Nachdem `gdb` an den Prozess attached hat, kann die Endlosschleife durch folgenden Befehl beendet werden:

```
set debugStop=false;
```

Bei der Übersetzung sollten folgende Optionen verwendet werden:

```
mpicxx -o <program name> -g -O0 <program source>
```

da der Compiler die Variabel ohne die Option `-O0` weg optimiert.

```
#!/bin/sh
echo "Running␣GDB␣on␣node␣‘hostname‘"
dir=‘pwd‘
if [ -e gdbfile ] ; then
  xterm -e "gdb␣-x␣$dir/gdbfile␣--args␣$*␣"
else
  xterm -e "gdb␣--args␣$*␣"
fi
exit 0
```

Speichert man dieses BASH-Skript in einer Datei `run_gdb.sh` und verwendet es in der Form

```
mpirun -np 4 run_gdb.sh myprogram
```

dann wird eine entsprechende Anzahl Fenster gestartet, in denen jeweils ein `gdb` Prozess mit dem Programm `myprogram` läuft.

Existiert im gleichen Verzeichnis eine Datei `gdbfile` dann werden aus dieser Argumente für das Programm gelesen.

Parallele Debugger

- Parallele Debugger stellen eine graphische Benutzeroberfläche für diesen Prozess zur Verfügung.

  Parallele Debugger sind z.B.

- Das Eclipse Plugin PTP.

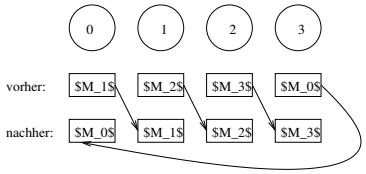
- Kommerzielle parallele Debugger, z.B. Totalview oder DDT.

11 Parallele Programmierung mit Message Passing: Fortsetzung

11.1 Vermeidung von Deadlocks: Färbung

Lokaler Austausch: Schieben im Ring

Betrachte folgendes Problem: Jeder Prozess $p \in \{0, \ldots, P-1\}$ muss ein Datum an $(p+1)\%P$ schicken:



Naives Vorgehen mit synchroner Kommunikation liefert Deadlock:

$$\ldots \ \mathbf{send}(\Pi_{(p+1)\%P}, msg); \ \ \mathbf{recv}(\Pi_{(p+P-1)\%P}, msg); \ \ \ldots$$

Aufbrechen des Deadlock (z. B. Vertauschen von **send**/**recv** in einem Prozess) liefert nicht die maximal mögliche Parallelität.

Asynchrone Kommunikation möchte man aus Effizienzgründen oft nicht verwenden.

Färben

Lösung: *Färben.* Sei $G = (V, E)$ ein Graph mit

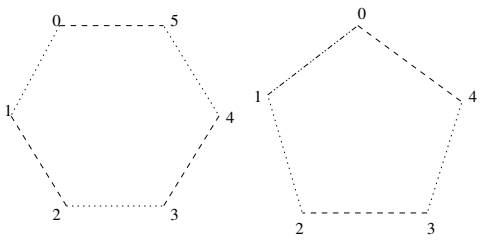
$$\begin{aligned} V &= \{0, \ldots, P-1\} \\ E &= \{e = (p, q) | \text{Prozess } p \text{ muß mit Prozess } q \text{ kommunizieren}\} \end{aligned}$$

Es sind die *Kanten* so einzufärben, dass an einem Knoten nur Kanten unterschiedlicher Farben anliegen. Die Zuordnung der Farben sei durch die Abbildung

$$c\colon E \to \{0, \ldots, C-1\}$$

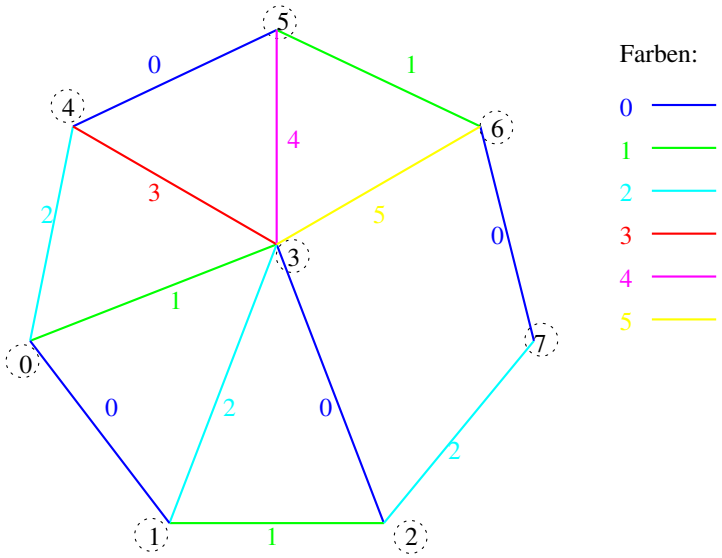
gegeben, wobei C die Zahl der benötigten Farben ist.

Schieben im *Ring* braucht zwei Farben für gerades P und drei Farben für ungerades P:



Lokaler Austausch: allgemeiner Graph

Ergeben die Kommunikationsbeziehungen einen allgemeinen Graph, so ist die Färbung algorithmisch zu bestimmen.



Hier eine mehr oder weniger sequentielle Heuristik:

Programm 11.1 (Verteiltes Färben).

```
parallel coloring
{
    const int P;
    process Π[int p ∈ {0,...,P − 1}]{
        int nbs;                                  // Anzahl Nachbarn
        int nb[nbs];                              // nb[i] < nb[i + 1] !
        int color[nbs];                           // das Ergebnis
        int index[MAXCOLORS];                     // Verwaltung der freien Farben
        int i, c, d;
        for (i = 0; i < nbs; i + +) index[i]=-1;
        for (i = 0; i < nbs; i + +)               // finde Farbe zur Verbindung zu nb[i]
            c = 0;                                // beginne mit Farbe 0
            while(1) {
                c=min{k ≥ c|index[k] < 0};        // nächste freie Farbe ≥ c
                if (p < nb[i]) { send(Π_nb[i],c); recv(Π_nb[i],d); }
                else { recv(Π_nb[i],c); send(Π_nb[i],d); }
                if (c == d){                      // die beiden haben sich geeinigt
                    index[c] = i; color[i] = c; break;
                } else c = max(c, d);
            }
        }
    }
}
```

11.2 Verteilter wechselseitiger Ausschluss

11.2.1 Lamport Zeitmarken

Benannt nach Leslie Lamport (Turing-Preis 2014, auch Schöpfer von LaTeX)

Ziel: Ordnen von Ereignissen in verteilten Systemen.

Ereignisse: Ausführen von (gekennzeichneten) Anweisungen.

Ideal wäre eine globale Uhr, die gibt es aber in verteilten Systemen nicht, da das Senden von Nachrichten immer mit Verzögerungen verbunden ist.

Logische Uhr: Den Ereignissen zugeordnete Zeitpunkte sollen nicht in offensichtlichem Widerspruch zu einer globalen Uhr stehen.

Π_1:	Π_2:	Π_3:
$a = 5;$		
$\ldots;$	$\ldots;$	$\vdots$
$b = 3;$	$c = 4;$	
$\mathbf{send}(\Pi_2, a);$	$\ldots;$	
	$\mathbf{recv}(\Pi_1, b);$	$e = 7;$
	$d = 8;$	$\mathbf{send}(\Pi_2, e);$
$\vdots$	$\mathbf{recv}(\Pi_3, e);$	
	$f = bde;$	
	$\vdots$	
	$\mathbf{send}(\Pi_1, f);$	
$\mathbf{recv}(\Pi_2, f);$		

Sei a ein Ereigniss in Prozess p und $C_p(a)$ die Zeitmarke, die Prozess p damit assoziiert, z. B. $C_2(f = bde)$. Dann sollen diese Zeitmarken folgende Eigenschaften haben:

1. Seien a und b zwei Ereignisse im selben Prozess p, wobei a vor b stattfinde, so soll $C_p(a) < C_p(b)$ gelten.
2. Es sende p eine Nachricht an q, so soll $C_p(\mathbf{send}) < C_q(\mathbf{receive})$ sein.
3. Für zwei beliebige Ereignisse a und b in beliebigen Prozessen p bzw. q gelte $C_p(a) \neq C_q(b)$.

1 und 2 spiegeln die Kausalität von Ereignissen wieder: Wenn in einem parallelen Programm sicher gesagt werden kann, dass a in p *vor* b in q stattfindet, dann gilt auch $C_p(a) < C_q(b)$.

Nur mit den Eigenschaften 1 und 2 wäre $a \leq_C b :\iff C_p(a) < C_q(b)$ eine Halbordnung auf der Menge aller Ereignisse.

Die Bedingung 3 macht daraus dann eine totale Ordnung.

Lamport Zeitmarken: Implementierung

Programm 11.2 (Lamport-Zeitmarken)**.**

```
parallel Lamport-timestamps
{
    const int P;                    // was wohl?
```

```
    int d = min{i|2^i ≥ P};              // wieviele Bitstellen hat P.
    process Π[int p ∈ {0,...,P − 1}]
    {
        int C=0;                         // die Uhr
        int t, s, r;                     // nur für das Beispiel
        int Lclock(int c)                // Ausgabe einer neuen Zeitmarke
        {
            C=max(C, c/2^d);             // Regel 2
            C++;                         // Regel 1
            return C · 2^d + p;          // Regel 3
                   // die letzten d Bits enthalten p
        }

        // Verwendung:
        // Ein lokales Ereignis passiert
        t=Lclock(0);
        // send
        s=Lclock(0);
        send(Π_q,message,s);             // Die Zeitmarke wird mit verschickt!
        // receive
        recv(Π_q,message,r);             // empfängt auch die Zeitmarke des Senders!
        r=Lclock(r);                     // so gilt C_p(r) > C_q(s)!
    }
}
```

Verwaltung der Zeitmarken obliegt dem Benutzer. Üblicherweise benötigt man Zeitmarken nur für ganz bestimmte Ereignisse (siehe unten).

Überlauf des Zählers wurde nicht behandelt.

Verwaltung der Zeitmarken obliegt dem Benutzer. Üblicherweise benötigt man Zeitmarken nur für ganz bestimmte Ereignisse (siehe unten).

Überlauf des Zählers wurde nicht behandelt.

Verteilter wechselseitiger Ausschluss mit Zeitmarken

Problem: Von einer Menge verteilter Prozesse soll genau einer etwas tun (z. B. ein Gerät steuern, als Server dienen, ...). Wie bei einem kritischen Abschnitt müssen sich die Prozesse einigen wer drankommt.

Eine Möglichkeit wäre natürlich, dass es einen Prozess gibt, der entscheidet wer drankommt.

Wir stellen jetzt eine alternative, verteilte Lösung vor:

- Will ein Prozess eintreten schickt er eine Nachricht an alle anderen.
- Sobald er Antwort von allen bekommen hat (es gibt kein Nein!) kann er eintreten.

- Ein Prozess bestätigt nur, wenn er nicht rein will oder wenn die Zeitmarke seines Eintrittswunsches größer ist als die des anderen.

Lösung arbeitet mit lokalem Monitorprozess.

Programm 11.3 (Verteilter wechselseitiger Ausschluss mit Lamport-Zeitmarken).

```
parallel DME-timestamp // Distributed Mutual Exclusion
{
    int P;                                  // ...
    const int REQUEST=1, REPLY=2;           // Nachrichten

    process Π[int p ∈ {0,...,P−1}]
    {
        int C=0, mytime;                    // Uhr
        int is_requesting=0, reply_pending;
        int reply_deferred[P]={0,...,0};    // abgewiesene Prozesse


    }
```

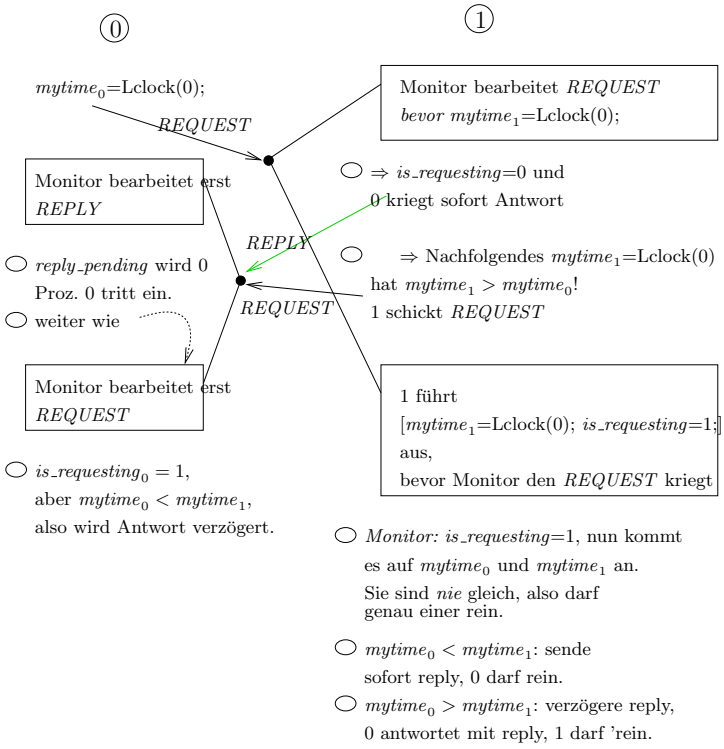
Programm 11.4 (Verteilter wechselseitiger Ausschluss mit Lamport-Zeitmarken).

```
parallel DME-timestamp (cont.)
{
    process M[int p‘ = p]                   // der Monitor
    {
        int msg, time;
        while(1) {
            recv_any(π,q,msg,time);         // Empfange von q's Monitor mit Zeitmarke des Senders
            if (msg==REQUEST)               // q will eintreten
            {
                [Lclock(time);]             // Erhöhe eigene Uhr für spätere Anfragen.
                                            // Kritischer Abschnitt, da Π auch erhöht.
                if(is_requesting ∧ mytime < time)
                    reply_deferred[q]=1;    // q soll warten
                else
                    asend(M_q,p,REPLY,0);// q darf 'rein
            }
            else
                reply_pending−−;            // es war ein REPLY
        }
    }

}
```

Programm 11.5 (Verteilter wechselseitiger Ausschluss mit Lamport-Zeitmarken).

```
parallel DME-timestamp (cont.)
{
```

```
    void enter_cs()                          // zum Eintreten in den kritischen Abschnitt
    {
        int i;
        [ mytime=Lclock(0); is_requesting=1; ]
                                             // kritischer Abschnitt
        reply_pending=P − 1;                 // so viele Antworten erwarte ich
        for (i=0; i < P; i++)
            if (i ≠ p) send(M_i,p,REQUEST,mytime);
        while (reply_pending> 0);            // busy wait
    }
    void leave_cs()
    {
        int i;
        is_requesting=0;
        for (i=0; i < P; i++)                // benachrichtig wartende Prozesse
            if (reply_deferred[i])
            {
                send(M_i,p,REPLY,0);
                reply_deferred[i]=0;
            }
    }

    enter_cs(); /* critical section */ leave_cs();
    } // end process
}
```



11.3 Wählen

Obiger Algorithmus braucht $2P$ Nachrichten pro Prozess um in den kritischen Abschnitt einzutreten. Beim Wählen werden wir mit $O(\sqrt{P})$ auskommen.

Insbesondere muss ein Prozess nicht *alle* anderen fragen bevor er rein darf.

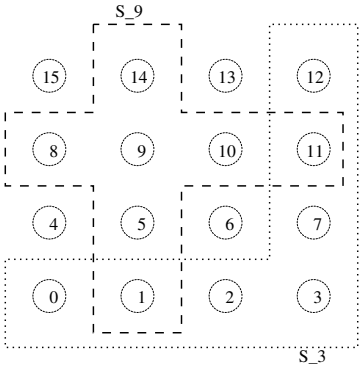
Idee:

- Die entsprechenden Prozesse bewerben sich um das Eintreten in den kritischen Abschnitt. Diese heißen *Kandidaten*
- Alle (oder einige, s. u.) stimmen darüber ab wer eintreten darf. Diese heißen *Wähler*. Jeder kann Kandidat und Wähler sein.
- Statt absoluter Mehrheit verlangen wir nur relative Mehrheiten: Ein Prozess darf eintreten sobald er weiß, dass kein anderer mehr Stimmen haben kann als er.

Jedem Prozess wird ein Wahlbezirk $S_p \subseteq \{0, \ldots, P-1\}$ zugeordnet. Es gelte die Überdeckungseigenschaft:

$$S_p \cap S_q \neq \emptyset \quad \forall p, q \in \{0, \ldots, P-1\}.$$

Die Wahlbezirke für Prozess 3 und 9 sehen bei 16 Prozessen so aus:



Ein Prozess p kann eintreten, wenn er alle Stimmen seines Wahlbezirkes bekommt. Denn kein anderer Prozess q kann eintreten: Nach Vor. existiert $r \in S_p \cap S_q$ und r hat sich für p entschieden, somit kann q nicht alle Stimmen haben.

Deadlockgefahr: Ist $|S_p \cap S_q| > 1$ so kann sich einer für p und ein anderer für q entscheiden, beide kommen dann nie dran. Lösen des Deadlocks mit Lamport Zeitmarken.

11.3.1 Verteilte Philosophen

Wir betrachten noch einmal das Philosophenproblem, diesmal jedoch mit message passing.

- Lasse eine Marke im Ring herumlaufen. Nur wer die Marke hat, darf eventuell essen.
- Zustandsänderungen werden dem Nachbarn mitgeteilt, **bevor** die Marke weitergeschickt wird.
- Jedem Philospohen P_i ist ein Diener W_i zugeordnet, der die Zustandsmanipulation vornimmt.

- Wir verwenden nur synchrone Kommunikation

```
process P_i [ int i ∈ {0,...,P−1}]
{
    while (1) {
        think;
        send(W_i, HUNGRY );
        recv( W_i, msg );
        eat;
        send( W_i, THINK );

    }
}

process W_i [ int i ∈ {0,...,P−1}]
{
    int L = (i+1)%P;
    int R = (i+p−1)%P ;
    int state = stateL = stateR = THINK ;
    int stateTemp;
    if ( i == 0 ) send( W_L , TOKEN );
    while (1) {
        recv_any( who, tag );
        if ( who == P_i ) stateTemp = tag ;                    // Mein Philosoph
        if ( who == W_L & & tag ≠ TOKEN ) stateL = tag ;       // Zustandsänderung
        if ( who == W_R & & tag ≠ TOKEN ) stateR = tag ;       // beim Nachbarn
        if ( tag == TOKEN){
            if ( state ≠ EAT & & stateTemp == HUNGRY
                & & stateL == THINK & & stateR == THINK ){
                    state = EAT;
                    send( W_l , EAT );
                    send( W_R , EAT );
                    send( P_i , EAT );
            }
            if ( state == EAT & & stateTemp == THINK ){
                state = THINK;
                send( W_L , THINK );
                send( W_R , THINK );
            }
            send( W_L , TOKEN );
        }
    }
}
```

12 Bewertung paralleler Algorithmen

Wie analysiert man die Eigenschaften eines parallelen Algorithmus zur Lösung eines Problems $\Pi(N)$?

- Problemgröße N ist frei wählbar
- Lösung des Problems mit sequentiellem bzw. parallelem Algorithmus
- Hardwarevoraussetzungen:
  - MIMD-Parallelrechner mit P identischen Rechenknoten
  - Kommunikationsnetzwerk skaliert mit der Anzahl der Rechenknoten
  - Aufsetzzeit, Bandbreite und Knotenleistung sind bekannt
- Ausführung des sequentiellen Programms auf einem Knoten
- Paralleler Algorithmus + Parallele Implementierung + Parallele Hardware = Paralleles System
- Der Begriff der Skalierbarkeit charakterisiert die Fähigkeit eines Parallelen Systems wachsende Resourcen in Form von Prozessoren P oder der Problemgröße N zu nutzen.

Ziel: Analyse der Skalierbarkeitseigenschaften eines Parallelen Systems
Maße für parallele Algorithmen

- Laufzeit
- Speedup und Effizienz
- Kosten
- Parallelitätsgrad

Definition verschiedener Ausführungszeiten:

- Die **sequentielle Ausführungszeit** $T_S(N)$ bezeichnet die Laufzeit eines sequentiellen Algorithmus zur Lösung des Problems Π bei Eingabegröße N.
- Die **optimale Ausführungszeit** $T_{best}(N)$ charakterisiert die Laufzeit des *besten* (existierenden) sequentiellen Algorithmus zur Lösung des Problems Π bei Eingabegröße N. Dieser hat für fast alle Größen von N den geringsten Zeitbedarf hat.
- Die **parallele Laufzeit** $T_P(N, P)$ beschreibt die Laufzeit des zu untersuchenden parallelen Systems zur Lösung von Π in Abhängigkeit der Eingabegröße N *und* der Prozessorzahl P.

12.1 Kenngrößen

Die Messung dieser Laufzeiten ermöglicht die Definition weiterer Größen:

- **Speedup**

$$S(N,P) = \frac{T_{best}(N)}{T_P(N,P)}. \tag{1}$$

Für alle N und P gilt $S(N,P) \leq P$. Angenommen es sei $S(N,P) > P$, dann existiert ein sequentielles Programm, welches das parallele Programm simuliert (im Zeitscheibenbetrieb abarbeitet). Dieses hypothetische Programm hätte dann die Laufzeit $PT_P(N,P)$ und es wäre

$$PT_P(N,P) = P\frac{T_{best}(N)}{S(N,P)} < T_{best}(N), \tag{2}$$

was offensichtlich ein Widerspruch ist.

- **Effizienz**

$$E(N,P) = \frac{T_{best}(N)}{PT_P(N,P)}. \tag{3}$$

Es gilt $E(N,P) \leq 1$. Die Effizienz gibt den Anteil des maximal erreichten Speedups an. Man kann auch sagen, daß $E \cdot P$ Prozessoren wirklich an der Lösung von Π arbeiten und der Rest $(1-E)P$ nicht effektiv zur Problemlösung beiträgt.

- **Kosten**

Als Kosten C definiert man das Produkt

$$C(N,P) = PT_P(N,P), \tag{4}$$

da man diese Rechenzeit im Rechenzentrum bezahlen müßte. Man bezeichnet einen Algorithmus als *kostenoptimal*, falls $C(N,P) = const T_{best}(N)$.

Offensichtlich gilt dann

$$E(N,P) = \frac{T_{best}(N)}{C(N,P)} = 1/const, \tag{5}$$

die Effizienz bleibt also konstant.

- **Parallelitätsgrad**

Mit $\Gamma(N)$ bezeichnen wir den *Parallelitätsgrad*. Das ist die maximale Zahl gleichzeitig ausführbarer Operationen im besten sequentiellen Algorithmus.

  - Offensichtlich könnten prinzipiell umso mehr Operationen parallel ausgeführt werden, je mehr Operationen überhaupt auszuführen sind, je größer also N ist. Der Parallelitätsgrad ist also von N abhängig.
  - Andererseits kann der Parallelitätsgrad nicht größer sein als die Zahl der insgesamt auszuführenden Operationen. Da diese Zahl proportional zu $T_S(N)$ ist, können wir sagen, dass

$$\Gamma(N) \leq O(T_S(N)) \tag{6}$$

gilt.

12.2 Speedup

Wesentlich ist das Verhalten des Speedups $S(N, P)$ eines parallelen Systems in Abhängigkeit von P.

Mit dem zweiten Parameter N hat man die Wahl verschiedener Szenarien.

1. Feste sequentielle Ausführungszeit

- Wir bestimmen N aus der Beziehung

$$T_{best}(N) \stackrel{!}{=} T_{fix} \quad \rightarrow \quad N = N_A \tag{7}$$

  wobei T_{fix} ein Parameter ist. Der skalierte Speedup ist dann

$$S_A(P) = S(N_A, P), \tag{8}$$

  dabei steht A für den Namen *Amdahl*.

- Wie verhält sich der skalierte Speedup? Annahme: das parallele Programm entsteht aus dem besten sequentiellen Programm mit sequentiellem Anteil $0 < q < 1$ und einem vollständig parallelisiertem Rest $(1-q)$. Die parallele Laufzeit (für das feste N_A!) ist also

$$T_P = qT_{fix} + (1-q)T_{fix}/P. \tag{9}$$

Für den Speedup gilt dann

$$S(P) = \frac{T_{fix}}{qT_{fix} + (1-q)T_{fix}/P} = \frac{1}{q + \frac{1-q}{P}} \tag{10}$$

Somit gilt das Gesetz von Amdahl

$$\lim_{P\to\infty} S(P) = 1/q. \tag{11}$$

Konsequenzen:

- Der maximal erreichbare Speedup wird also rein durch den sequentiellen Anteil bestimmt.
- Die Effizienz sinkt stark ab wenn man nahe an den maximalen Speedup herankommen will.
- Diese Erkenntnis führte Ende der 60er Jahre zu einer sehr pessimistischen Einschätzung der Möglichkeiten des parallelen Rechnens.
- Dies hat sich erst geändert als man erkannt hat, dass für die meisten parallelen Algorithmen der sequentielle Anteil q mit steigendem N *abnimmt*.

Der Ausweg aus dem Dilemma besteht also darin mit mehr Prozessoren immer größere Probleme zu lösen!

Wir stellen nun drei Ansätze vor wie man N mit P wachsen lassen kann.

2. Feste parallele Ausführungszeit

Wir bestimmen N aus der Gleichung

$$T_P(N, P) \stackrel{!}{=} T_{fix} \quad \rightarrow \quad N = N_G(P) \tag{12}$$

für gegebenes T_{fix} und betrachten dann den Speedup

$$S_G(P) = S(N_G(P), P). \tag{13}$$

- Diese Art der Skalierung wird auch „Gustafson Skalierung“ genannt.
- Motivation sind bspw. Anwendungen im Bereich der Wettervorhersage. Hier hat man eine bestimmte Zeit T_{fix} zur Verfügung in der man ein möglichst großes Problem lösen will.

3. Fester Speicherbedarf pro Prozessor

Viele Simulationsanwendungen sind speicherbeschränkt, der Speicherbedarf wächst als Funktion $M(N)$. Je nach Speicherkomplexität bestimmt nicht die Rechenzeit sondern der Speicherbedarf welche Probleme man auf einer Maschine noch berechnen kann.

Annahme: Nehmen wir an, dass der Parallelrecher aus P identischen Prozessoren besteht, die jeweils einen Speicher der Grösse M_0 besitzen, so bietet sich die Skalierung

$$M(N) \stackrel{!}{=} PM_0 \quad \rightarrow \quad N = N_M(P) \tag{14}$$

an und wir betrachten

$$S_M(P) = S(N_M(P), P). \tag{15}$$

als skalierten Speedup.

4. Konstante Effizienz

Wir wählen N so, dass die parallele Effizienz konstant bleibt.

Wir fordern

$$E(N, P) \stackrel{!}{=} E_0 \quad \rightarrow \quad N = N_I(P). \tag{16}$$

Dies bezeichnet man als *isoeffiziente Skalierung*. Offensichtlich ist $E(N_I(P), P) = E_0$ also

$$S_I(P) = S(N_I(P), P) = PE_0. \tag{17}$$

Eine isoeffiziente Skalierung ist nicht für alle parallelen Systeme möglich. Man findet nicht unbedingt eine Funktion $N_I(P)$, die (16) identisch erfüllt. Somit kann man andererseits vereinbaren, dass ein System skalierbar ist genau dann wenn eine solche Funktion gefunden werden kann.

12.3 Beispiel

Zur Vertiefung der Begriffe betrachten wir ein **Beispiel**

- Es sollen N Zahlen auf einem Hypercube mit P Prozessoren addiert werden. Wir gehen folgendermassen vor:
  - Jeder hat N/P Zahlen, die er im ersten Schritt addiert.
  - Diese P Zwischenergebnisse werden dann im Baum addiert.

- Wir erhalten somit die sequentielle Rechenzeit

$$T_{best}(N) = (N-1)t_a \tag{18}$$

- die parallele Rechenzeit beträgt

$$T_P(N,P) = (N/P-1)t_a + \operatorname{ld} P t_m, \tag{19}$$

wobei t_a die Zeit für die Addition zweier Zahlen und t_m die Zeit für den Nachrichtenaustausch ist (wir nehmen an, dass $t_m \gg t_a$).

1. Feste sequentielle Ausführungszeit (Amdahl)
Setzen wir $T_{best}(N) = T_{fix}$ so erhalten wir, falls $T_{fix} \gg t_a$, in guter Näherung

$$N_A = T_{fix}/t_a.$$

Für sinnvolle Prozessorzahlen P gilt: $P \leq N_A$. Für den Speedup erhalten wir im Fall von $N_A/P \gg 1$

$$S_A(P) = \frac{T_{fix}}{T_{fix}/P + \operatorname{ld} P t_m} = \frac{P}{1 + P \operatorname{ld} P \frac{t_m}{T_{fix}}}. \tag{20}$$

2. Feste parallele Ausführungszeit (Gustafson)
Hier erhält man

$$\left(\frac{N}{P} - 1\right) t_a + \operatorname{ld} P t_m = T_{fix} \Longrightarrow N_G = P\left(1 + \frac{T_{fix} - \operatorname{ld} P t_m}{t_a}\right). \tag{21}$$

Die maximal nutzbare Prozessorzahl ist wieder beschränkt: $2^{T_{fix}/t_m}$. Ist $\operatorname{ld} P t_m = T_{fix}$, so wird bei ein Einsatz von mehr Prozessoren in jedem Fall die maximal zulässige Rechenzeit überschritten. Immerhin können wir annehmen, dass $2^{T_{fix}/t_m} \gg T_{fix}/t_a$ gilt.

Der skalierte Speedup S_G ist unter der Annahme $N_G(P)/P \gg 1$:

$$S_G(P) = \frac{N_G(P) t_a}{N_G(P) t_a/P + \operatorname{ld} P t_m} = \frac{P}{1 + \operatorname{ld} P \frac{t_m}{T_{fix}}}. \tag{22}$$

Es gilt $N_G(P) \approx P T_{fix}/t_a$. Für gleiche Prozessorzahlen ist also S_G größer als S_A.

3. Fester Speicher pro Prozessor (Speicherlimitierung)
Der Speicherbedarf ist $M(N) = N$, damit gilt für $M(N) = M_0 P$ die Skalierung

$$N_M(P) = M_0 P.$$

Wir können nun unbegrenzt viele Prozessoren einsetzen, im Gegenzug steigt die parallele Rechenzeit auch unbegrenzt an. Für den skalierten Speedup bekommen wir:

$$S_M(P) = \frac{N_M(P) t_a}{N_M(P) t_a/P + \operatorname{ld} P t_m} = \frac{P}{1 + \operatorname{ld} P \frac{t_m}{M_0 t_a}}. \tag{23}$$

Für die Wahl $T_{fix} = M_0 t_a$ ist dies die selbe Formel wie S_G. In beiden Fällen sehen wir, dass die Effizienz mit P fällt.

4. Isoeffiziente Skalierung
Wir wählen N so, dass die Effizienz konstant bleibt, bzw. der Speedup linear wächst, d.h.:

$$S = \frac{P}{1 + \frac{P \operatorname{ld} P}{N} \frac{t_m}{t_a}} \stackrel{!}{=} \frac{P}{1+K} \Longrightarrow N_I(P) = P \operatorname{ld} P \frac{t_m}{K t_a},$$

für ein frei wählbares $K > 0$. Da $N_I(P)$ existiert wird man den Algorithmus als skalierbar bezeichnen. Für den Speedup gilt $S_I = P/(1+K)$.

12.4 Isoeffizienzanalyse

Weitere Formalisierung des Prinzips der isoeffizienten Skalierung

- Ziel Beantwortung von Fragestellungen: „Ist dieser Algorithmus zur Matrixmultiplikation auf dem Hypercube besser skalierbar als jener zur schnellen Fouriertransformation auf einem Feld“
- Problemgröße: Parameter N war bisher willkürlich gewählt.
- N bei der Matrixmultiplikation kann sowohl Anzahl der Matrixelemente als auch Elementanzahl pro Zeile bezeichnen.
- In diesem Fall würde sich im ersten Fall $2N^{3/2}t_f$ und im zweiten Fall $2N^3t_f$ als sequentielle Laufzeit ergeben.
- Vergleichbarkeit von Algorithmen erfordert Invarianz des Aufwandsmaß bezüglich der Wahl des Problemgrößenparameters.
- Wir wählen als Maß für den Aufwand W eines (sequentiellen) Algorithmus dessen Ausführungszeit, wir setzen also

  $$W = T_{best}(N) \tag{24}$$

  selbst. Diese Ausführungszeit ist wiederum proportional zur Zahl der auszuführenden Operationen in dem Algorithmus.
- Für den Parallelitätsgrad Γ erhält man:

  $$\Gamma(W) \leq O(W),$$

  denn es können nicht mehr Operationen parallel ausgeführt werden als überhaupt auszuführen sind.
- Mittels $N = T_{best}^{-1}(W)$ können wir schreiben

  $$\tilde{T}_P(W, P) = T_P(T_{best}^{-1}(W), P),$$

  wobei wir jedoch die Tilde im folgenden gleich wieder weglassen werden.

Wir definieren den *Overhead* als

$$T_o(W, P) = PT_P(W, P) - W \geq 0. \tag{25}$$

$PT_P(W, P)$ ist die Zeit, die eine Simulation des sequentiellen Programms auf einem Prozessor benötigen würde. Diese ist in jedem Fall nicht kleiner als die beste sequentielle Ausführungszeit W. Der Overhead beinhaltet zusätzliche Rechenzeit aufgrund von Kommunikation, Lastungleichheit und „überflüssigen“ Berechnungen.

Isoeffizienzfunktion Aus dem Overhead erhalten wir

$$T_P(W, P) = \frac{W + T_o(W, P)}{P}.$$

Also erhalten wir für den Speedup

$$S(W,P) = \frac{W}{T_P(W,P)} = P\frac{1}{1+\frac{T_o(W,P)}{W}},$$

bzw. für die Effizienz

$$E(W,P) = \frac{1}{1+\frac{T_o(W,P)}{W}}.$$

Im Sinne einer isoeffizienten Skalierung fragen wir nun: Wie muss W als Funktion von P wachsen damit die Effizienz konstant bleibt. Wegen obiger Formel ist dies dann der Fall wenn $T_o(W,P)/W = K$, mit einer beliebigen Konstanten $K \geq 0$. Die Effizienz ist dann $1/(1+K)$.

- Eine Funktion $W_K(P)$ heißt *Isoeffizienzfunktion* falls sie die Gleichung

  $$T_o(W_K(P),P) = KW_K(P)$$

  identisch erfüllt.

- Ein paralleles System heißt skalierbar genau dann wenn es eine Isoeffizienzfunktion besitzt.

- Das asymptotische Wachstum von W mit P ist ein Maß für die Skalierbarkeit eines Systems: Besitzt etwa ein System S_1 eine Isoeffizienzfunktion $W = O(P^{3/2})$ und ein System S_2 eine Isoeffizienzfunktion $W = O(P^2)$ so ist S_2 *schlechter* skalierbar als S_1.

Wann gibt es eine Isoeffizienzfunktion?

- Wir gehen aus von der Effizienz

  $$E(W,P) = \frac{1}{1+\frac{T_o(W,P)}{W}}.$$

- Effizienz bei *festem* W und wachsendem P. Es gilt für jedes parallele System, dass

  $$\lim_{P\to\infty} E(W,P) = 0$$

  wie man aus folgender Überlegung sieht: Da W fest ist, ist auch der Parallelitätsgrad fest und damit gibt es eine untere Schranke für die parallele Rechenzeit: $T_P(W,P) \geq T_{min}(W)$, d.h. die Berechnung kann nicht schneller als T_{min} sein, egal wieviele Prozessoren eingesetzt werden. Damit gilt jedoch asymptotisch

  $$\frac{T_o(W,P)}{W} \geq \frac{PT_{min}(W) - W}{W} = O(P)$$

  und somit geht die Effizienz gegen 0.

Betrachten wir nun die Effizienz bei *festem* P und wachsender Arbeit W, so gilt für viele (nicht alle!) parallele Systeme, dass

$$\lim_{W\to\infty} E(W,P) = 1.$$

Offensichtlich bedeutet dies im Hinblick auf die Effizienzformel, dass

$$T_o(W,P)|_{P=const} < O(W) \tag{26}$$

bei festem P wächst also der Overhead weniger als linear mit W. In diesem Fall kann für jedes P ein W gefunden werden so dass eine gewünschte Effizienz erreicht wird. Gleichung (26) sichert also die Existenz einer Isoeffizienzfunktion. Als Beispiel für ein nicht skalierbares System sei die Matrixtransposition genannt. Wir werden später ableiten, dass der Overhead in diesem Fall $T_o(W,P) = O(W, \mathrm{ld}\, P)$ beträgt. Somit existiert keine Isoeffizienzfunktion.

Optimal parallelisierbare Systeme

Wir wollen nun der Frage nachgehen wie Isoeffizienzfunktionen mindestens wachsen müssen. Dazu bemerken wir zunächst, dass

$$T_P(W,P) \geq \frac{W}{\Gamma(W)},$$

denn $\Gamma(W)$ (dimensionslos) ist ja die maximale Zahl gleichzeitig ausführbarer Operationen im sequentiellen Algorithmus bei Aufwand W. Somit ist $W/\Gamma(W)$ eine untere Schranke für die parallele Rechenzeit.

Nun können sicher nicht mehr Operationen parallel ausgeführt werden als überhaupt auszuführen sind, d.h. es gilt $\Gamma(W) \leq O(W)$. Wir wollen ein System als *optimal parallelisierbar* bezeichnen falls

$$\Gamma(W) = cW$$

gilt mit einer Konstanten $c > 0$. Nun gilt

$$T_P(W,P) \geq \frac{W}{\Gamma(W)} = \frac{1}{c},$$

die minimale parallele Rechenzeit bleibt konstant. Für den Overhead erhalten wir dann in diesem Fall

$$T_o(W,P) = PT_P(W,P) - W = P/c - W$$

und somit für die Isoeffizienzfunktion

$$T_o(W,P) = P/c - W \stackrel{!}{=} KW \Longleftrightarrow W = \frac{P}{(K+1)c} = \Theta(P).$$

Optimal parallelisierbare Systeme haben somit eine Isoeffizienzfunktion $W = \Theta(P)$. Wir merken uns also, dass Isoeffizienzfunktionen mindestens linear in P wachsen.

Wir werden in den folgenden Kapiteln die Isoeffizienzfunktionen für eine Reihe von Algorithmen bestimmen, deswegen sei hier auf ein ausführliches Beispiel verzichtet.

13 MPI-2

Einige hilfreiche MPI-1 Funktionen

```
1 double MPI_Wtime();
```

Die Funktion `MPI_Wtime` liefert die Sekunden seit einem definierten Zeitpunkt in der Vergangenheit zurück. Sinnvoll sind daher nur Differenzen von Zeiten zwischen zwei verschiedenen Aufrufen von `MPI_Wtime`.

```
double MPI_Wtick();
```

Die Funktion `MPI_Wtick` liefert die Auflösung der für `MPI_Wtime` verwendeten Uhr in Sekunden zurück.

```
int MPI_Abort(MPI_Comm comm, int error_code);
```

Der Aufruf von `MPI_Abort` führt zum Abbruch aller am Kommunikator `comm` beteiligten Prozesse. Die Prozesse liefern den Fehlercode `error_code` zurück (ebenso wie wenn in `main` ein `return error_code` ausgeführt worden wäre).

Der MPI-2 Standard ist eine Erweiterung von MPI-1, das weiter eine vollständige Teilmenge von MPI-2 bildet. Eine Änderung an existierenden Programmen ist daher nicht notwendig.

Die Änderungen betreffen insbesondere

- Die dynamische Erzeugung von Prozessen
- Erweiterte Kommunikation über Interkommunikatoren
- Hybrider Parallelismus (Threads)
- Einseitige Kommunikation
- Paralleler Dateizugriff

13.1 Dynamische Prozessverwaltung

- Im MPI-1 Standard wird davon ausgegangen, dass die Anzahl Prozesse beim Start des parallelen Programms festgelegt wird und sich während des Programmablaufs nicht ändert.
- MPI-2 ermöglicht dynamisches Erzeugen von Prozessen.
- Dies hat Folgen für die Kommunikatoren. Während `MPI_COMM_WORLD` bei MPI-1 immer alle Prozesse enthielt, ist das bei dynamischer Prozesserzeugung nicht mehr so einfach möglich. `MPI_COMM_WORLD` enthält daher zunächst die initial gestarteten Prozesse.
- Die neu erzeugten Kind-Prozesse besitzen ihren eigenen Kommunikator `MPI_COMM_WORLD`. Dieser ermöglicht nur die Kommunikation zwischen den in einem Schritt erzeugten Kind-Prozessen.
- Die Funktion `int MPI_Comm_get_parent(MPI_Comm *parent)` liefert einen Interkommunikator, über den mit den Elternprozessen kommuniziert werden kann. Diesen Interkommunikator erhalten die Eltern auch von der Funktion, die die Kind-Prozesse erzeugt.

Prozesserzeugung

```
int MPI_Comm_spawn(char *program, char *argv[], int numprocs,
2         MPI_Info info, int root, MPI_Comm comm, MPI_Comm
             *intercomm,
          int errcodes[])
```

- Mit dieser Funktion werden `numprocs` neue Prozesse erzeugt, die das Programm `program` ausführen.
- Man kann Ihnen einen Argumentvektor übergeben. Dabei ist zu beachten, dass der Argumentvektor für `MPI_Comm_spawn` nicht wie gewohnt im Element 0 den Programmnamen enthält, dieser wird automatisch hinzugefügt. Es werden also direkt Argumente angegeben, die der erzeugte Prozess dann ab Element 1 findet. Das letzte Argument muss ein Nullpointer sein. Alternativ kann `MPI_ARGV_NULL` verwendet werden.
- Mit der `MPI_Info`-Struktur könnten Werte an die Laufzeitumgebung übergeben werden. Dies führt jedoch zu MPI-Programmen die nicht portabel sind. Es ist deshalb sinnvoll hier i.d.R. `MPI_INFO_NULL` anzugeben.
- Nur der Prozess mit dem Rank `root` erzeugt tatsächlich die neuen Prozesse.
- `MPI_Comm_spawn` ist eine kollektive Operation. Alle am Kommunikator `comm` beteiligten Prozesse müssen Sie aufrufen.
- `intercomm` enthält anschließend einen Interkommunikator, mit dem ein Datenaustausch zwischen Vater- und Kind-Prozessen möglich ist.
- `errcodes` liefert mögliche Fehlercodes der erzeugten Programme zurück
- `program, argv, numprocs` und `info` werden nur beim Prozess mit Rank `root` ausgewertet.

```
int MPI_Comm_spawn_multiple(int count, char *program[],
2        char **argv[], int numprocs[], MPI_Info info[], int root,
         MPI_Comm comm, MPI_Comm *intercomm, int errcodes[])
```

- Im Prinzip wie `MPI_Comm_spawn` allerdings können gleichzeitig Prozesse gestartet werden, die `count` verschiedene Programme ausführen.
- Entsprechend sind `program, argv, numprocs` und `info` jetzt Arrays, die für jeden Programmtyp unterschiedliche Werte beinhalten können.
- Alle Prozesse gehören aber zum gleichen Interkommunikator `intercomm`. Das wäre anders, wenn die Prozesse mit mehreren Aufrufen von `MPI_Comm_spawn` erzeugt worden wären.
- Die Größe von `errcodes` muss $\sum_{i=0}^{\mathsf{count}} \mathsf{numprocs}[i]$ sein.

13.2 Interkommunikatoren

- Bei MPI-1 waren **Inter**kommunikatoren nur für den paarweisen Datenaustausch zwischen Prozessen verwendbar. Globale Kommunikation (Broadcast, Reduce, Barrier ...) gab es nur für **Intra**kommunikatoren.

- Als gemeinsame Kommunikationsoperationen für Interkommunikatoren gab es nur `MPI_Intercomm_create()` und `MPI_Comm_dup()` zum Erzeugen von und Klonen von Interkommunikatoren.

- In MPI-2 gibt es viele zusätzliche Kommunikationsfunktionen für Interkommunikatoren.

- Dies ist insbesondere im Hinblick auf die dynamische Prozesserzeugung wichtig.

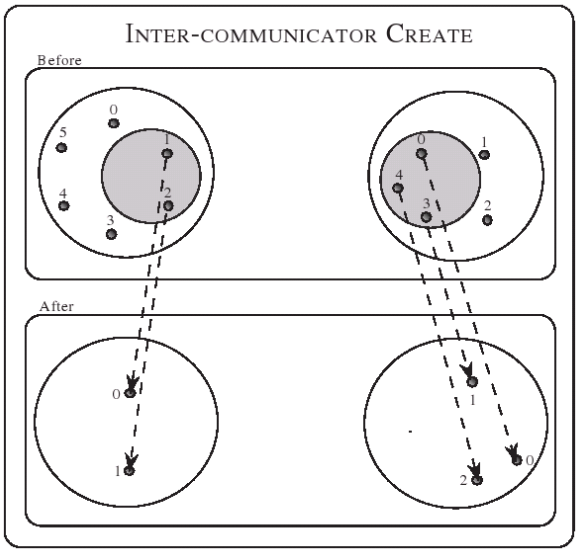
- Ein Interkommunikator verbindet zwei Gruppen von MPI-Prozessen. Insbesondere bei den kollektiven Operationen spielen dabei die Root-Prozesse der beiden Gruppen eine zentrale Rolle.

Erzeugen von Interkommunikatoren

```
int MPI_Intercomm_create(MPI_Comm local_comm, int local_leader,
                         MPI_Comm peer_comm, int remote_leader,
                         int tag, MPI_Comm *newintercomm)

int MPI_Comm_create(MPI_Comm comm, MPI_Group group,
                    MPI_Comm *newcomm)
```

- `MPI_Intercomm_create` erzeugt neuen Interkommunikator `newintercomm` aus den beiden Intrakommunikatoren `local_comm` und `peer_comm`, wobei die Prozesse mit dem Rank `local_leader` und `root_leader` zu den Root Prozessen der jeweiligen Gruppen werden. Diese Möglichkeit gab es schon in MPI-1.

- `MPI_Comm_create` erstellt einen neuen Intrakommunikator `newcomm` aus den Mitgliedern der Gruppe `group`, wenn `comm` ein Intrakommunikator ist. Neu ist bei MPI-2, dass dies auch für Interkommunikatoren funktioniert. Dort sind die Gruppen dann für jede Teilmenge des Interkommunikators verschieden. Ergebnis ist ein neuer Interkommunikator. Dies geht mit weniger Aufwand als die Erzeugung von neuen Kommunikatoren mit `MPI_Comm_split` und anschliessendes `MPI_Intercomm_create`.



aus dem MPI-2 Standard

Erzeugung eines neuen Interkommunikators aus zwei Teilgruppen eines bestehenden.

Weitere Funktionen für Interkommunikatoren

```
int MPI_Intercomm_merge(MPI_Comm intercomm, int high,
                        MPI_Comm *newintracomm)

int MPI_Comm_test_inter(MPI_Comm comm, int *flag)

int MPI_Comm_remote_size(MPI_Comm comm, int *size)

int MPI_Comm_remote_group(MPI_Comm comm, MPI_Group *group)
```

- `MPI_Intercomm_merge` erstellt einen neuen Intrakommunikator `newintracomm` aus den beiden Gruppen die über `intercomm` verbunden sind. Die Prozesse der Gruppe, bei der `high` 0 gesetzt ist, kommt im neuen Kommunikator vor denen der anderen Gruppe.
- `MPI_Comm_test_inter` überprüft ob der Kommunikator `comm` ein Interkommunikator ist und liefert das Ergebnis in `flag` zurück.
- `MPI_Comm_remote_size` liefert die Größe der nicht-lokalen Gruppe von `comm` in `size` zurück, `MPI_Comm_remote_group` erzeugt eine `MPI_Group` mit den Mitgliedern der nicht-lokalen Gruppe (entsprechendes liefern `MPI_Comm_size` und `MPI_Comm_group` für die lokale Gruppe).

MPI-Gruppen

- MPI-Gruppen sind eines der fundamentalen Konzepte von MPI. Sie fassen eine Menge von Prozessen zusammen, die dann einen konsekutiven Rank haben. Ein Kommunikator besteht immer aus einer MPI-Gruppe und einem Kontext. Ein Kontext kennzeichnet verschiedene Arten von möglicher Kommunikation, die sich nicht in die Quere kommen sollen.

- Für Operationen bei denen es einfach um Mengen von Prozessen geht, lassen sich MPI-Gruppen auch unabhängig von Kommunikatoren verwenden. Eine Gruppe hat den Typ `MPI_Group`
- Die zu einem Kommunikator `comm` gehörenden Gruppe `group` erhält man durch

```
int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)
```

```
int MPI_Group_union(MPI_Group group1, MPI_Group group2,
                    MPI_Group *newgroup)
int MPI_Group_intersection(MPI_Group group1, MPI_Group group2,
                           MPI_Group *newgroup)
int MPI_Group_difference(MPI_Group group1, MPI_Group group2,
                         MPI_Group *newgroup)
```

- Da Gruppen Prozessmengen sind, lässt sich eine neue Gruppe `newgroup` als die Vereinigungsmenge, Schnittmenge oder Differenzmenge zweier existierender Gruppen `group1` und `group2` bilden.

```
int MPI_Group_incl(MPI_Group group, int n, const int ranks[],
                   MPI_Group *newgroup)
int MPI_Group_excl(MPI_Group group, int n, const int ranks[],
                   MPI_Group *newgroup)
```

- Eine neue Gruppe `newgroup` lässt sich aus einer existierenden Gruppe `group` auch durch Auswahl oder durch Ausschluss von n Gruppenmitgliedern durch Angabe ihrer Ranks im Array `ranks` erzeugen. Es gibt auch noch Varianten mit denen gleich ganze Bereiche von Ranks ausgewählt werden können.

Weitere Funktionen für Gruppen

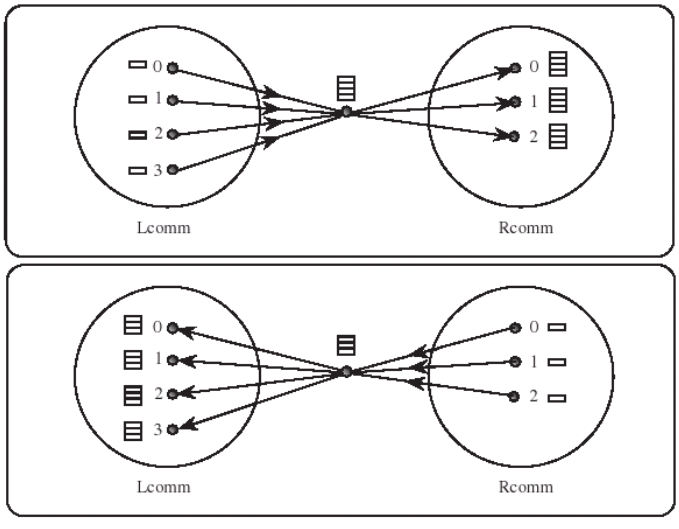
```
int MPI_Group_size(MPI_Group group, int *size)

int MPI_Group_rank(MPI_Group group, int *rank)

int MPI_Group_compare(MPI_Group group1, MPI_Group group2,
                      int *result)

int MPI_Group_translate_ranks(MPI_Group group1, int n,
        const int ranks1[], MPI_Group group2, int ranks2[])
```

- Es ist natürlich möglich die Anzahl von Prozessen in einer Gruppe, sowie den Rank des aktuellen Prozesses in dieser Gruppe herauszufinden. Gehört der aufrufende Prozess nicht zu der Gruppe, wird `MPI_UNDEFINED` zurückgeliefert.
- Zwei Gruppen lassen sich auf Gleichheit prüfen.
- Es ist auch möglich den Rank von n Prozessen `ranks1` der Gruppe `group1` in der Gruppe `group2` zu erfahren.

Kollektivkommunikation für Interkommunikatoren

Es gibt eine ganze Reihe von Kollektivoperationen nicht nur für Intra- sondern auch für Interkommunikatoren.

- Alle an Alle
  - `MPI_Allgather, MPI_Allgatherv`
  - `MPI_Alltoall, MPI_Alltoallv`
  - `MPI_Allreduce, MPI_Reduce_scatter`
- Alle an Einen
  - `MPI_Gather, MPI_Gatherv`
  - `MPI_Reduce`
- Einer an Alle
  - `MPI_Bcast`
  - `MPI_Scatter, MPI_Scatterv`
- Andere
  - `MPI_Barrier`
- Bei einer Barrier auf einem Interkommunikator ist das Verhalten genau wie erwartet, alle Prozesse beider Teilgruppen warten, bis alle Prozesse angekommen sind.
- Die kollektive Kommunikation läuft immer von der Quellgruppe zur Zielgruppe. Bei Einer an Alle und Alle an Einen Operationen erfolgt die Kommunikation zwischen dem Rootprozess der Quell- bzw. Zielgruppe und den Prozessen der Ziel- bzw. Quellgruppe.
- Bei Alle an alle Operationen wird erst in der einen Richtung über den Rootprozess der einen Gruppe und dann in der anderen Richtung über den Rootprozess der anderen Gruppe kommuniziert.



aus dem MPI-2 Standard

Nachrichten und Datenfluß bei `MPI_Allgather()`

Reduce für Intrakommunikatoren

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

- Die Syntax ist (aus Gründen der Kompatibilität) im wesentlichen die gleiche wie für das Reduce bei Intrakommunikatoren.
- Die sendende Gruppe gibt bei root den Rank des Rootprozesses der empfangenden Gruppe an. Bei der empfangenden Gruppe setzt der Rootprozess für `root` die Konstante `MPI_ROOT` ein und die anderen Prozesse `MPI_PROC_NULL`.
- Das Ergebnis der Reduktion landet dann im Rootprozess der empfangenden Gruppe.

Beispielprogramm

```
#include "mpi.h"
#include <cstring>
#include <iostream>
#include <sstream>

const int NUM_SPAWNS=4;

int main( int argc, char *argv[] )
{
  bool parent=true;
  if ((argc>1) && strcmp(argv[1],"child")==0)
    parent=false;
  MPI_Init(&argc, &argv);
  int globalRank;
  MPI_Comm_rank(MPI_COMM_WORLD,&globalRank);

  if (parent)
  {
    char **argvC = new char*[2];
    argvC[0] = new char[128];
    strcpy(argvC[0],"child");
    argvC[1] = 0;
    MPI_Comm intercomm;
    int errcodes[NUM_SPAWNS];
    MPI_Comm_spawn(argv[0], argvC, NUM_SPAWNS, MPI_INFO_NULL,
                   0, MPI_COMM_WORLD, &intercomm, errcodes);
    int localRank,localSize,value=42+globalRank;
    MPI_Comm_rank(intercomm,&localRank);
    MPI_Comm_size(intercomm,&localSize);
    if (globalRank==1)
      MPI_Bcast(&value,1,MPI_INT,MPI_ROOT,intercomm);
```

```
    else
      MPI_Bcast(&value,1,MPI_INT,MPI_PROC_NULL,intercomm);
    MPI_Barrier(intercomm);
    std::ostringstream buffer;
    buffer << "I'm␣the␣parent␣" << localRank << "␣of␣"
           << localSize << "␣and␣send␣" << value << ".\n";
    std::cout << buffer.str();

    if (globalRank==0) // reduce from child group to parent root
    {
      MPI_Reduce(0,&value,1,MPI_INT,MPI_SUM,MPI_ROOT,intercomm);
      buffer.str("");
      buffer << "I'm␣the␣parent␣" << localRank << "␣of␣"
             << localSize << "␣and␣reduce␣was␣" << value <<
                ".\n";
      std::cout << buffer.str();
    }
    else
      MPI_Reduce(0,&value,1,MPI_INT,MPI_SUM,MPI_PROC_NULL,intercomm);
    MPI_Barrier(MPI_COMM_WORLD); // only parent processes
    MPI_Reduce(&value,0,1,MPI_INT,MPI_SUM,0,intercomm);
    delete [] argvC[0];
    delete [] argvC;
  }
  else
  {
    MPI_Comm parentcomm;
    MPI_Comm_get_parent(&parentcomm);
    int localRank,localSize,value;
    MPI_Comm_rank(MPI_COMM_WORLD,&localRank);
    MPI_Comm_size(MPI_COMM_WORLD,&localSize);
    MPI_Bcast(&value,1,MPI_INT,1,parentcomm);
    MPI_Barrier(parentcomm);     // child and parent processes
    std::ostringstream buffer;

    buffer << "I'm␣the␣spawned␣" << localRank << "␣of␣"
           << localSize << "␣and␣received␣" << value << ".\n";
    std::cout << buffer.str();
    MPI_Reduce(&value,0,1,MPI_INT,MPI_SUM,0,parentcomm);
    if (globalRank==0)  // reduce from parent group to child
       root
    {
      MPI_Reduce(0,&value,1,MPI_INT,MPI_SUM,MPI_ROOT,parentcomm);
      buffer.str("");
      buffer << "I'm␣the␣parent␣" << localRank << "␣of␣"
             << localSize << "␣and␣reduce␣was␣" << value <<
                ".\n";
```

```
11        std::cout << buffer.str();
      }
13      else
        MPI_Reduce(0,&value,1,MPI_INT,MPI_SUM,MPI_PROC_NULL,parentcomm);
15      MPI_Barrier(MPI_COMM_WORLD); // only child processes
    }
17    MPI_Finalize();
    return 0;
19  }
```

Ausgabe

```
I'm the parent 0 of 4 and send 42.
2 I'm the parent 0 of 4 and reduce was 172.
I'm the parent 2 of 4 and send 44.
4 I'm the spawned 0 of 4 and received 43.
I'm the spawned 1 of 4 and received 43.
6 I'm the spawned 2 of 4 and received 43.
I'm the spawned 3 of 4 and received 43.
8 I'm the parent 1 of 4 and send 43.
I'm the parent 3 of 4 and send 45.
10 I'm the parent 0 of 4 and reduce was 304.
```

13.3 Hybride Parallelisierung mit MPI und Threads

Grundsätzliche Annahmen

- Es gibt eine Möglichkeit zur Erzeugung von Threads nach POSIX Standard (OpenMP, Pthreads, C++-Threads...)
- Ein MPI Prozess kann uneingeschränkt multi-threaded ablaufen
- Multithreading sollte nicht mit der dynamischen Prozesserzeugung verwechselt werden. Bei letzterer entstehen echte eigene Prozesse. Beim Multithreading nur mehrere Threads für einen Prozess.
- Jeder Thread könnte MPI Funktionen aufrufen. Dies kann allerdings eingeschränkt werden.
- Die Threads eines MPI Prozesses sind nicht unterscheidbar Ein Rank spezifiziert einen MPI Prozess nicht einen seiner Threads
- Der Benutzer muss race conditions verhindern, welche durch widersprüchliche Kommunikationsaufrufe entstehen können. Dies könnte z.B. durch thread-spezfifische Kommunikatoren geschehen

Minimale Anforderungen an ein Thread-verträgliches MPI

- Alle MPI Aufrufe sind threadsafe, d.h. zwei nebenläufige Threads dürfen MPI Aufrufe absetzen, das Ergebnis ist invariant bezüglich der Aufrufreihenfolge, auch bei zeitlicher Verschränkung der Aufrufe.

- Blockierende MPI Aufrufe blockieren nur den aufrufenden Thread, während weitere Threads aktiv sein können, insbesondere können diese MPI Aufrufe ausführen.
- MPI Aufrufe kann man threadsafe machen indem man zu einem Zeitpunkt nur einen Aufruf ausführt. Dies kann man mit einem MPI Prozess eigenem Lock bewerkstelligen.

Initialisierung

- Sollen Programme multi-threaded ablaufen, so wird statt `MPI_Init()` die Funktion

```
int MPI_Init_thread(int *argc, char ***argv, int required,
2                   int *provided)
```

  aufgerufen.

- Dabei ist `required` der Wunsch des Ausmaßes der Unterstützung für Multithreading:
  - `MPI_THREAD_SINGLE`: es gibt kein Multithreading.
  - `MPI_THREAD_FUNNELED`: der Prozess kann multi-threaded sein, aber alle MPI Aufrufe werden ausschließlich vom Hauptthread gemacht.
  - `MPI_THREAD_SERIALIZED`: der Prozess kann multi-threaded sein und alle Threads dürfen MPI Aufrufe ausführen, aber zu einem Zeitpunkt nur einer (also keine Nebenläufigkeit von MPI Aufrufen)
  - `MPI_THREAD_MULTIPLE`: Alle Threads dürfen MPI ohne Einschränkungen aufrufen.
- In `provided` wird zurückgeliefert, welcher Threadlevel tatsächlich unterstützt wird. Dies kann weniger sein, als gewünscht. Insbesondere unterstützten nicht alle MPI-Implementierungen `MPI_THREAD_MULTIPLE`.
- Der Benutzer hat dafür zu sorgen, dass die Beschränkungen des Threadlevels eingehalten werden.
- Es ist nicht garantiert, dass eine eventuelle Ausnahmenbehandlung vom gleichen Thread bewerkstelligt wird, welcher den MPI Aufruf, der die Ausnahmebehandlung verursacht hat, ausgeführt hat.

```
int MPI_Query_thread(int *provided)
2
int MPI_Is_thread_main(int *flag)
```

- Mit `MPI_Query_thread` lässt sich auch während des Programms das Ausmaß der Threadunterstützung abfragen.
- `MPI_Is_thread_main` liefert in `flag` true zurück, wenn der aufrufenden Thread der Hauptthread ist, also der Thread der auf jeden Fall kommunizieren darf.

Beispielprogramm

```
#include "mpi.h"
#include "omp.h"
#include <iostream>
#include <sstream>

int main( int argc, char *argv[] )
{
  int provided;
  MPI_Init_thread(&argc, &argv, MPI_THREAD_FUNNELED, &provided);
  std::ostringstream buffer;
  buffer << "Thread level required: " << MPI_THREAD_FUNNELED
         << " thread level provided: " << provided << std::endl;
  std::cout << buffer.str();

  omp_set_num_threads(4);

  #pragma omp parallel private(buffer)
  {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    int mainThread=0;
    MPI_Is_thread_main(&mainThread);
    if (mainThread)
    {
      buffer.str("");
      buffer << "I'm the main thread of rank " << rank <<
          std::endl;
      std::cout << buffer.str();
    }
    else
    {
      buffer.str("");
      buffer << "I'm a spawned thread of rank " << rank <<
          std::endl;
      std::cout << buffer.str();
    }
  }
  MPI_Finalize();
  return 0;
}
```

Ausgabe

```
Thread level required: 1 thread level provided: 1
Thread level required: 1 thread level provided: 1
I'm the main thread of rank 1
```

```
I'm a spawned thread of rank 1
I'm the main thread of rank 0
I'm a spawned thread of rank 0
I'm a spawned thread of rank 1
I'm a spawned thread of rank 1
I'm a spawned thread of rank 0
I'm a spawned thread of rank 0
```

13.4 Einseitige Kommunikation

- Einseitige Kommunikation ist eine Erweiterung des Kommunikationsmechanismus um Remote Memory Access (RMA)
- Im Prinzip gibt es drei Funktionen: `MPI_Put`, `MPI_Get` und `MPI_Accumulate`.
- Damit die Funktionen aufgerufen werden können muss der andere Prozess erst mit `MPI_Win_create` ein Speicherfenster vom Typ `MPI_Win` öffnen, aus dem gelesen oder in das geschrieben werden darf. Mit `MPI_Win_free` wird dieses wieder geschlossen.
- Zusätzlich gibt es verschiedene Funktionen zur globalen oder lokalen Synchronisation oder für eine Art kritischen Abschnitt.
- Vorteil: Nutzung von Architekturmerkmalen (gemeinsamer Speicher, Hardware-unterstützte Put/Get Operationen, DMA-Zugriff)

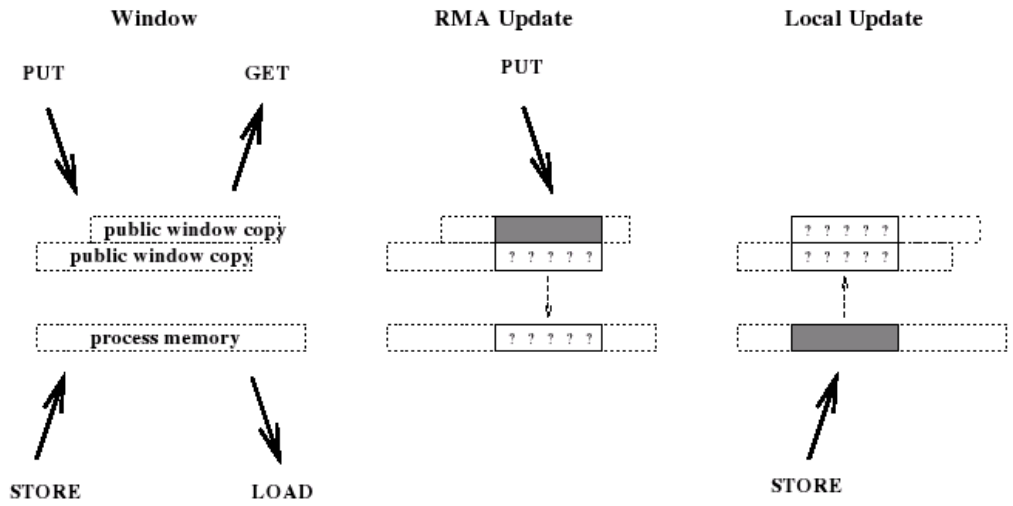
```
int MPI_Win_create(void *base, MPI_Aint size, int disp_unit,
                   MPI_Info info, MPI_Comm comm, MPI_Win *win)

int MPI_Win_free(MPI_Win *win)
```

- `base` gibt die Basisadresse des Speicherfensters an, `disp_unit` die Größe eines Datenelements. `info` könnte für Informationen an die Laufzeitumgebung verwendet werden, auch hier ist `MPI_INFO_NULL` zu empfehlen.
- `comm` ist der Kommunikator der beteiligten Prozesse. `MPI_Win_create` ist eine kollektive Operation, die von allen Prozessen des Kommunikators aufgerufen werden muss.
- Auf das erzeugte Fenster kann dann über `win` zugegriffen werden.

```
int MPI_Put(const void *origin_addr, int origin_count,
            MPI_Datatype origin_datatype, int target_rank,
            MPI_Aint target_disp, int target_count,
            MPI_Datatype target_datatype, MPI_Win win)
int MPI_Get(void *origin_addr, int origin_count,
            MPI_Datatype origin_datatype, int target_rank,
            MPI_Aint target_disp, int target_count,
            MPI_Datatype target_datatype, MPI_Win win)
int MPI_Accumulate(const void *origin_addr, int origin_count,
```

```
10          MPI_Datatype origin_datatype, int target_rank,
            MPI_Aint target_disp, int target_count,
12          MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)
```

- `origin_addr` ist die Basisadresse des Speicherfensters aus dem lokal gelesen oder geschrieben wird. `origin_count` gibt die Anzahl der Elemente und `origin_datatype` deren Typ an.
- `target_rank` ist natürlich der Rank des Zielprozesses.
- `target_disp` ein Offset in Anzahl Elementen im Speicherfenster des Zielprozesses.
- `target_count` ist die Anzahl der zu schreibenden Elemente und `target_datatype` deren Typ. Dies ist sicher oft gleich, muss aber nicht sein.
- `win` ist das Speicherfenster.



aus dem MPI-2 Standard

Verschiedene Varianten der Implementierung einseitigen Zugriffs.

Synchronisation

```
int MPI_Win_fence(int assert, MPI_Win win)
```

- Um sicherzustellen, dass die Kommunikationsoperationen auch abgeschlossen sind, ist eine Synchronisierung notwendig.
- Eine kollektive Variante der Synchronisierung ist `MPI_Win_fence`. Es wird nur beendet, wenn alle gestarteten einseitigen Kommunikationsoperationen für das Speicherfenster `win` abgeschlossen sind.
- Mit `assert` lassen sich zusätzliche Überprüfungen veranlassen. Ein Wert von 0 ist hier meistens ausreichend.

```
int MPI_Win_start(MPI_Group exposegroup, int assert, MPI_Win
    win)
int MPI_Win_complete(MPI_Win win)

int MPI_Win_post(MPI_Group accessgroup, int assert, MPI_Win win)
int MPI_Win_wait(MPI_Win win)
```

- Mit diesen Funktionen sind lokale Synchronisationen möglich. Dabei greifen die Prozesse, die `MPI_Win_start` aufgerufen haben, lesend oder schreibend auf das Speicherfenster `win` der Prozesse der Gruppe `exposegroup` zu.
- Diese müssen dies den entsprechenden Prozessen in der Gruppe `accessgroup` durch Aufruf von `MPI_Win_post` erlauben.
- Der Zugriff wird vor Beenden von `MPI_Win_complete` und `MPI_Win_wait` abgeschlossen.
- Es gibt mit `MPI_Win_test` auch eine nicht-blockierende Variante von `MPI_Win_wait`.
- Mit `MPI_Win_lock` gibt es auch die Möglichkeit nur einem einzelnen Rank Zugriff zu gewähren.

Beispielprogramm

```
#include "mpi.h"
#include <cstring>
#include <iostream>
#include <array>

const int NUM_SPAWNS=4;

int main( int argc, char *argv[] )
{
  MPI_Init(&argc, &argv);
  int globalRank;
  MPI_Comm_rank(MPI_COMM_WORLD,&globalRank);
  std::array<int,32> dataStorage;
  for (auto &value : dataStorage)
    value = 0;

  MPI_Win win; // Create window for one-sided communication
  MPI_Win_create(&dataStorage[0], 32*sizeof(int), sizeof(int),
                 MPI_INFO_NULL, MPI_COMM_WORLD, &win);

  if (globalRank==0)  // fill some fields and send them to the
     other process
  {
    for (int i=0;i<16;++i)
      dataStorage[i]=i;
```

```
5       MPI_Put(&dataStorage[0],16,MPI_INT,1,0,16,MPI_INT,win);
    }
7   MPI_Win_fence(0,win);  // wait till all data transfer is
       finished
    if (globalRank==1)  // do something with the data
9   {
      for (int i=0;i<16;++i)
11      dataStorage[i+16]=2*dataStorage[i];
    }
13  // create sender and receiver groups for one-sided
       communication
    MPI_Group worldGroup;
15  MPI_Comm_group(MPI_COMM_WORLD, &worldGroup);
    MPI_Group accessGroup;
17  int ranks[1] = {0};
    MPI_Group_incl(worldGroup,1,ranks,&accessGroup);

1   MPI_Group exposeGroup;
    ranks[0] = 1;
3   MPI_Group_incl(worldGroup,1,ranks,&exposeGroup);
    if (globalRank==0)  // rank 0 wants to receive data
5   {
      MPI_Win_start(exposeGroup,0,win);
7     MPI_Get(&dataStorage[16],16,MPI_INT,1,16,16,MPI_INT,win);
      MPI_Win_complete(win);
9     for (int i=0;i<32;++i)
        std::cout << dataStorage[i] << "␣";
11    std::cout << std::endl;
    }
13  else if (globalRank==1)  // rank 1 allows the access to data
    {
15    MPI_Win_post(accessGroup,0,win);
      MPI_Win_wait(win);
17  }
    MPI_Win_free(&win);  // the window is no longer needed
19  MPI_Finalize();
    return 0;
21 }
```

Ausgabe

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 2 4 6 8 10 12 14 16 18
   20 22 24 26 28 30
```

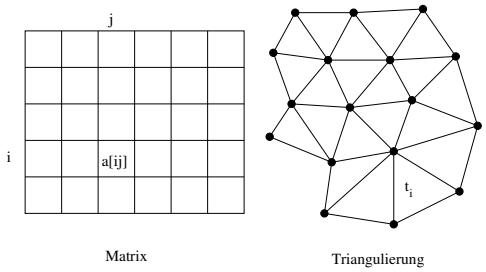
14 Grundlagen paralleler Algorithmen

Parallelisierungsansätze zum Entwurf paralleler Algorithmen:

1. Partitionierung: *Zerlegen* eines Problems in unabhängige Teilaufgaben. Dies dient der Identifikation des maximal möglichen Parallelismus.
2. Agglomeration: Kontrolle der *Granularität* um Rechenaufwand und Kommunikation auszubalancieren.
3. Mapping: Abbilden der Prozesse auf Prozessoren. Ziel ist eine optimale Abstimmung der logischen Kommunikationsstruktur mit der Maschinenstruktur.

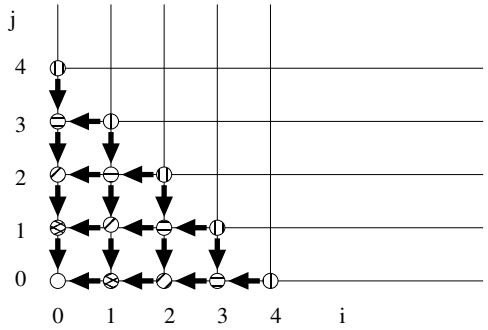
14.1 Partitionierung

- Berechnungen sind direkt an eine bestimmte Datenstrukturen geknüpft.
- Für jedes Datenobjekt sind gewisse Operationen auszuführen, oftmals ist dieselbe Folge von Operationen auf unterschiedliche Daten pro Objekt anzuwenden. Somit kann jedem Prozess ein Teil der Daten(objekte) zugeordnet werden.



- Matrixaddition $C = A + B$: Es können alle Elemente c_{ij} vollkommen parallel bearbeitet werden. In diesem Fall würde man jedem Prozess Π_{ij} die Matrixelemente a_{ij}, b_{ij} und c_{ij} zuordnen.
- Triangulierung bei numerischer Lösung partieller Differentialgleichungen: Hier treten Berechnungen pro Dreieck auf, die alle gleichzeitig durchgeführt werden können, jedem Prozess würde man somit eine Teilmenge der Dreiecke zuordnen.

Datenabhängigkeit



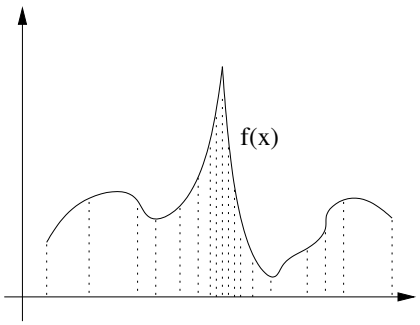
Datenabhängigkeiten im Gauß–Seidel–Verfahren.

- Operationen können oft nicht für alle Datenobjekte gleichzeitig durchgeführt werden.
- Beispiel: Gauß-Seidel-Iteration mit lexikographischer Nummerierung. Berechnung am Gitterpunkt (i, j) hängt vom Ergebnis der Berechnungen an den Gitterpunkten $(i-1, j)$ und $(i, j-1)$ ab.
- Der Gitterpunkt $(0, 0)$ kann ohne jede Voraussetzung berechnet werden.
- Alle Gitterpunkte auf den Diagonalen $i + j = const$ können parallel bearbeitet werden.

Funktionale Zerlegung

- bei unterschiedlichen Operationen auf gleichen Daten.
- Beispiel Compiler: Dieser vollführt die Schritte lexikalische Analyse, Parsing, Codegenerierung, Optimierung und Assemblierung. Jeder Schritt kann einem separaten Prozess zugeordnet werden („Makropipelining“).

Irreguläre Probleme



- Keine a-priori Zerlegung möglich
- Berechnung des Integrals einer Funktion $f(x)$ durch adaptive Quadratur
- Intervallbreite hängt von f ab und ergibt sich während der Berechnung

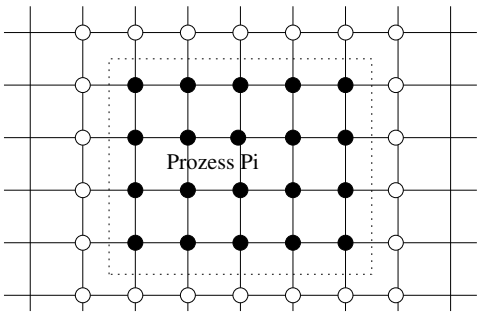
14.2 Agglomeration

- Zerlegungsschritt zeigt die maximale Parallelität auf.
- Nutzung (im Sinne von ein Datenobjekt pro Prozess) ist meist nicht sinnvoll (Kommunikationsaufwand)
- Agglomeration: Zuordnung von mehreren Teilaufgaben zu einem Prozess, damit wird Kommunikation für diese Teilaufgaben in möglichst wenigen Nachrichten zusammengefaßt.
- Reduktion der Anzahl der zu sendenden Nachrichten, weitere Einsparungen bei *Datenlokalität*

- Als *Granularität* eines parallelen Algorithmus bezeichnet man das Verhältnis:

$$\text{Granularität} = \frac{\text{Anzahl Nachrichten}}{\text{Rechenzeit}}.$$

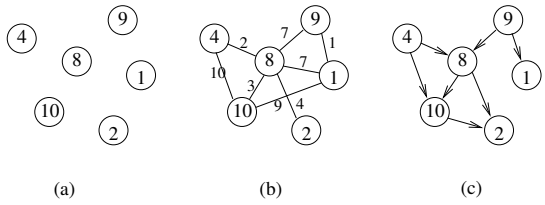
- Agglomeration reduziert also die Granularität.



Gitterbasierte Berechnungen

- Berechnungen werden für alle Gitterpunkte parallel durchgeführt werden.
- Zuweisung einer Menge von Gitterpunkten zu einem Prozeß
- Alle Modifikationen auf Gitterpunkten können u. U. gleichzeitig durchgeführt, es bestehen also keine Datenabhängigkeiten.
- Ein Prozess besitze N Gitterpunkte und muss somit $\mathcal{O}(N)$ Rechenoperationen ausführen.
- Nur für Gitterpunkte am Rand der Partition ist eine Kommunikation notwendig.
- Es ist also somit nur für insgesamt $4\sqrt{N}$ Gitterpunkte Kommunikation notwendig.
- Verhältnis von Kommunikationsaufwand zu Rechenaufwand verhält sich somit wie $\mathcal{O}(\sqrt{N})$ (bzw. $\mathcal{O}(N^{2/3})$ in 3D)
- Erhöhung der Anzahl der Gitterpunkte pro Prozessor macht den Aufwand für die Kommunikation relativ zur Rechnung beliebig klein *Oberfläche–zu–Volumen–Effekt.*

Wie ist die Agglomeration durchzuführen?



1. Ungekoppelte Berechnungen
2. Gekoppelte Berechnungen
3. Gekoppelte Berechnungen mit zeitlicher Abhängigkeit

Verteilen

(a) Ungekoppelte Berechnungen

- Berechnung besteht aus Teilproblemen, die völlig unabhängig voneinander berechnet werden können.
- Jedes Teilproblem kann unterschiedliche Rechenzeit benötigen.
- Darstellbar als Menge von Knoten mit Gewichten. Gewichte sind ein Maß für die erforderliche Rechenzeit.
- Agglomeration ist trivial. Man weist die Knoten der Reihe nach (z.B. der Größe nach geordnet oder zufällig) jeweils dem Prozess zu der am wenigsten Arbeit hat (dies ist die Summe all seiner Knotengewichte).
- Agglomeration wird komplizierter wenn die Anzahl der Knoten sich erst während der Berechnung ergibt (wie bei der adaptiven Quadratur) und/oder die Knotengewichte a priori nicht bekannt sind (wie z.B. bei depth first search).

→ Lösung durch dynamische Lastverteilung

- zentral: ein Prozess nimmt die Lastverteilung vor
- dezentral: ein Prozess holt sich Arbeit von anderen, die zuviel haben

(b) Gekoppelte Berechnungen

- Standardmodell für statische, datenlokale Berechnungen.
- Berechnung wird durch einen ungerichteten Graphen beschrieben.
- Erst ist eine Berechnung pro Knoten erforderlich, deren Berechnungsdauer wird vom Knotengewicht modelliert. Danach tauscht jeder Knoten Daten mit seinen Nachbarknoten aus.
- Anzahl der zu sendenden Daten ist proportional zum jeweiligen Kantengewicht.
- Regelmäßiger Graph mit konstanten Gewichten: triviale Agglomeration.
- Allgemeiner Graph $G = (V, E)$ bei P Prozessoren: Knotenmenge V ist so zu partititonieren, dass

$$\bigcup_{i=1}^{P} V_i = V, \quad V_i \cap V_j = \emptyset, \quad \sum_{v \in V_i} g(v) = \left(\sum_{v \in V} g(v) \right) / P$$

  und die Separatorkosten

$$\sum_{(v,w) \in S} g(v,w) \to \min, \; S = \{(v,w) \in E | v \in V_i, w \in V_j, i \neq j\}$$

  minimal sind.
- Dieses Problem wird als *Graphpartitionierungsproblem* bezeichnet.

- bereits im Fall konstanter Gewichte und $P = 2$ (Graphbisektion) $\mathcal{NP}$–vollständig.
- Es existieren gute Heuristiken, welche in linearer Zeit (in der Anzahl der Knoten, $\mathcal{O}(1)$ Nachbarn) eine (ausreichend) gute Partitionierung erzeugen.

(c)Gekoppelte Berechnungen mit zeitlicher Abhängigkeit

- Modell ist ein gerichteter Graph.
- Ein Knoten kann erst berechnet werden kann, wenn alle Knoten von eingehenden Kanten berechnet sind.
- Ist ein Knoten berechnet wird das Ergebnis über die ausgehenden Kanten weitergegeben. Die Rechenzeit entspricht den Knotengewichten, die Kommunikationszeit entspricht den Kantengewichten.
- Im allgemeinen sehr schwer lösbares Problem.
- Theoretisch nicht „schwieriger als Graphpartitionierung“ (auch $\mathcal{NP}$–vollständig)
- Praktisch sind keine einfachen und guten Heuristiken bekannt.
- Für spezielle Probleme, z.B.adaptive Mehrgitterverfahren, kann man jedoch gute Heuristiken finden.

14.3 Mapping

Abbilden der Prozesse auf Prozessoren:

- Menge der Prozesse Π bildet ungerichteten Graphen $G_\Pi = (\Pi, K)$: Zwei Prozesse sind miteinander verbunden, wenn sie miteinander kommunizieren (Kantengewichte könnten den Umfang der Kommunikation modellieren).
- Ebenso bildet die Menge der Prozessoren P mit dem Kommunikationsnetzwerk einen Graphen $G_P = (P, N)$: Hypercube, Feld.
- Sei $|\Pi| = |P|$ und es stellt sich folgende Frage: Welcher Prozess soll auf welchem Prozessor ausgeführt werden?
- Im allgemeinen wollen wir die Abbildung so durchführen, dass Prozesse, die miteinander kommunizieren möglichst auf benachbarte oder nahe Prozessoren abgebildet werden.
- Dieses Optimierungsproblem bezeichnet man als *Graphabbildungsproblem*
- Dieses ist (leider) wieder $\mathcal{NP}$–vollständig.
- Heutige Multiprozessoren besitzen sehr leistungsfähige Kommunikationsnetzwerke: In cut-through-Netzwerken ist, wie wir gesehen haben, die Übertragungszeit einer Nachricht praktisch entfernungsunabhängig.
- Das Problem der optimalen Prozessplatzierung ist damit nicht mehr vorrangig.
- Eine gute Prozessplatzierung ist dennoch wichtig falls viele Prozesse *gleichzeitig* miteinander kommunizieren wollen.

14.4 Lastverteilung

14.4.1 Statische Lastverteilung ungekoppelter Probleme

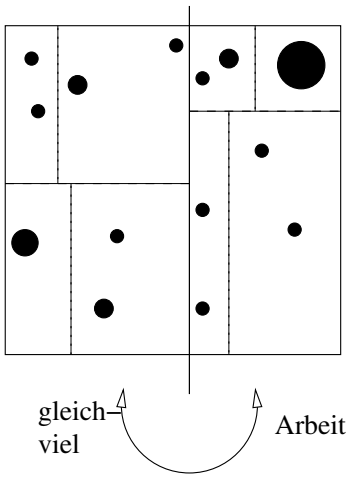
- Aufgabe: Aufteilung der anfallenden Arbeit auf die verschiedenen Prozessoren.

- Dies entspricht dem Agglomerationsschritt bei welchem man die parallel abarbeitbaren Teilprobleme wieder zusammenfasst.

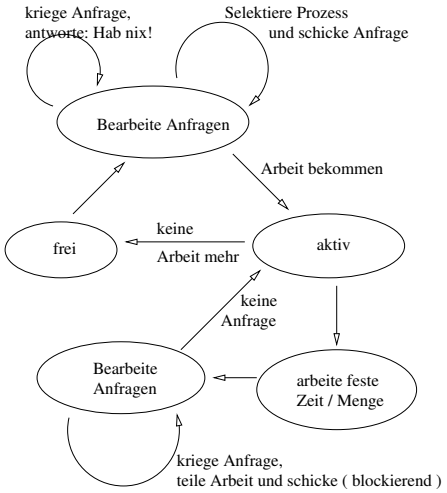
- Das Maß für die Arbeit sei dabei bekannt.

- **Bin Packing**
  - Anfangs sind alle Prozessoren leer.
  - Knoten, die in beliebiger Reihenfolge oder sortiert (z.B. der Größe nach) vorliegen, werden nacheinander auf den Prozessor mit der aktuell wenigsten Arbeit gepackt.
  - Das funktioniert auch dynamisch, falls im Rechenvorgang neue Arbeit entsteht.

- **Rekursive Bisektion**
  - Jedem Knoten sei eine Position im Raum zugeordnet.
  - Der Raum wird orthogonal zum Koordinatensystem so geteilt, dass in den Teilen etwa gleich viel Arbeit besteht.
  - Dieses Vorgehen wird dann rekursiv auf die entstandenen Teilräume mit alternierenden Koordinatenrichtungen angewandt.



14.4.2 Dynamische Lastverteilung ungekoppelter Probleme



Aktivitäts-/Zustandsdiagramm

- Das Maß für die Arbeit sei unbekannt.
- Prozess ist entweder aktiv (verrichtet Arbeit) oder frei (arbeitslos).
- Beim Teilen der Arbeit sind folgende Fragen zu berücksichtigen:
  - Was will ich abgeben? Beim Travelling-Salesman Problem z.B. möglichst Knoten aus dem Stack, die weit unten liegen.
  - Wieviel will ich abgeben? z.B. die Hälfte der Arbeit (half split).
- Neben der Arbeitsverteilung kann weitere Kommunikation stattfinden (Bildung des globalen Minimums bei branch-and-bound beim Travelling-Salesman).
- Des weiteren besteht das Problem der Terminierungserkennung hinzu.
  Wann sind alle Prozesse idle?

Welcher Idle-Prozess soll als nächster angesprochen werden?
Verschiedene Selektionsstrategien:

- **Master/Slave(Worker)Prinzip**

  Ein Prozess verteilt die Arbeit. Er weiß, wer aktiv bzw. frei ist und leitet die Anfrage weiter. Er regelt (da er weiß, wer frei ist) auch das Terminierungsproblem. Nachteil: diese Methode skaliert nicht. Alternativ: hierarchische Struktur von Mastern.

- **Asynchrones Round Robin**

  Der Prozess Π_i hat eine Variable target_i. Er schickt seine Anfrage an Π_{target_i} und setzt dann $\text{target}_i = (\text{target}_i + 1)\%P$.

- **Globales Round Robin**

  Es gibt nur eine globale Variable target. Vorteil: keine gleichzeitigen Anfragen an denselben Prozess. Nachteil: Zugriff auf eine globale Variable (was z.B. ein Server Prozess machen kann) .

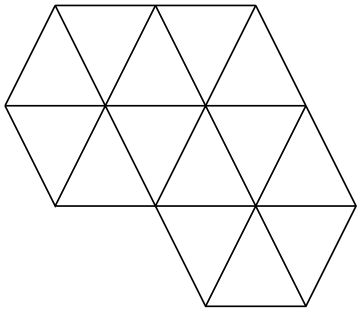
- **Random Polling**

  Jeder wählt zufällig einen Prozess mit gleicher Wahrscheinlichkeit ($\rightarrow$ Paralleler Zufallsgenerator, achte zumindest auf das Austeilen von seeds oder ähnlichem). Dieses Vorgehen bietet eine gleichmäßige Verteilung der Anfragen und benötigt keine globale Resource.

14.4.3 Graphpartitionierung für gekoppelte Probleme

Aufteilung von Finite-Element-Netzen

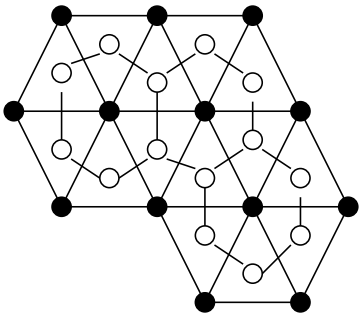
Betrachten wir ein Finite-Elemente-Netz:



Es besteht aus Dreiecken $T = \{t_1, \ldots, t_N\}$ mit

$$\bar{t_i} \cap \bar{t_j} = \begin{cases} \emptyset \\ \text{ein Knoten} \\ \text{eine Kante} \end{cases}$$

- Arbeit findet beim Verfahren der Finiten Elemente in den Knoten statt.
- Alternativ kann man auch in jedes Dreieck einen Knoten legen, diese mit Kanten verbinden und den so entstehenden Dualgraphen betrachten.



Graph und zugehöriger dualer Graph

Die Aufteilung des Graphen auf Prozessoren führt zum Graphpartitionierungsproblem. Dazu machen wir die folgenden Notationen:

$G = (V, E)$ (Graph oder Dualgraph)
$E \subseteq V \times V$ symmetrisch (ungerichtet)

Die Gewichtsfunktionen

$w : V \longrightarrow \mathbb{N}$ (Rechenaufwand)
$w : E \longrightarrow \mathbb{N}$ (Kommunikation)

Die Gesamtarbeit ist $W = \sum_{v \in V} w(v)$
Außerdem sei k die Anzahl der zu bildenden Partitionen, wobei $k \in \mathbb{N}$ und $k \geq 2$ sei. Gesucht ist nun eine Partitionsabbildung

$$\pi : V \longrightarrow \{0, \ldots, k-1\}$$

und der dazugehörige Kantenseparator

$$X_\pi := \{(v, v') \in E \mid \pi(v) \neq \pi(v')\} \subseteq E$$

Das Graphpartitionierungproblem besteht nun darin, die Abbildung π so zu finden, dass das Kostenfunktional (Kommunikationskosten)

$$\sum_{e \in X_\pi} w(e) \to \min$$

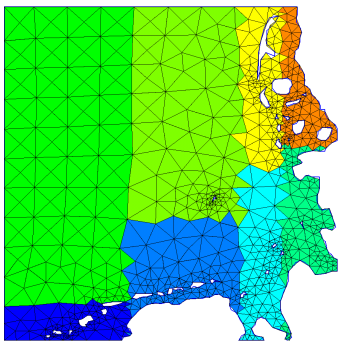
minimal wird unter der Nebenbedingung (Gleichverteilung der Arbeit)

$$\sum_{v,\pi(v)=i} w(v) \leq \delta \frac{W}{k} \quad \text{für alle } i \in \{0, \ldots, k-1\}$$

wobei δ das erlaubte Ungleichgewicht bestimmt ($\delta = 1.1 \iff 10\%$ Abweichung).

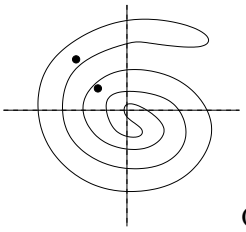
- Annahme: Berechnungkosten dominieren Kommunikationskosten. Ansonsten wäre wegen der hohen Kommunikationskosten evtl. die Partitionierung sinnlos. Das ist allerdings nur ein Modell für die Laufzeit!
- Bei Zweiteilung spricht man vom Graphbisektionsproblem. Durch rekursive Bisektion lassen sich 2^d-Wege-Partitionierungen erzeugen.
- Problematischerweise ist die Graphpartitionierung $\mathcal{NP}$-vollständig für k ≥ 2 .
- Eine optimale Lösung würde mehr Aufwand verursachen als die eigentliche Rechnung, der parallele Overhead wäre nicht akzeptabel.

$\to$ Notwendigkeit schneller Heuristiken.



Rekursive Koordinatenbisektion (RCB)

- Man benötigt die Positionen der Knoten im Raum (bei Finite-Elemente Anwendungen sind die vorhanden).
- Bisher haben wir das Verfahren unter dem Namen **Rekursive Bisektion** gesehen.
- Diesmal ist das Problem gekoppelt. Daher ist es wichtig, dass der Raum, in dessen Koordinaten die Bisektion durchgeführt wird, mit dem Raum, in dem die Knoten liegen, übereinstimmt.
- Im Bild ist das nicht der Fall. Zwei Knoten mögen zwar räumlich dicht beieinanderliegen, eine Koordinatenbisektion macht aber keinen Sinn, da die Punkte hier nicht gekoppelt sind, es also einem Prozessor gar nichts nützt, beide zu speichern.

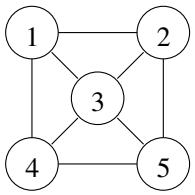


Gegenbeispiel zur Anwendbarkeit der RCB

Rekursive Spektralbisektion (RSB)

Hier werden die Positionen der Knoten im Raum nicht benötigt. Man stellt zunächt die Laplacematrix A(G) zum vorliegenden Graphen G auf. Diese ist folgendermaßen definiert:

$$A(G) = \{a_{ij} : 1 \leq i, j \leq |V|\} \quad \text{mit} \quad a_{ij} = \begin{cases} \text{Knotengrad}(v_i) & i = j \\ -1 & (v_i, v_j) \in E \\ 0 & \text{sonst} \end{cases}$$



	1	2	3	4	5
1	3	−1	−1	−1	0
2	−1	3	−1	0	−1
3	−1	−1	4	−1	−1
4	−1	0	−1	3	−1
5	0	−1	−1	−1	3

Graph und zugehörige Laplacematrix

Dann löse das Eigenwertproblem

$$Ae = \lambda e$$

Der kleinste Eigenwert λ_1 ist gleich Null, denn mit $e_1 = (1, \ldots, 1)^T$ gilt $Ae_1 = 0 \cdot e_1$. Der zweitkleinste Eigenwert λ_2 allerdings ist ungleich Null, falls der Graph zusammenhängend ist.

Die Bisektion findet nun anhand der Komponenten des Eigenvektors e_2 statt, und zwar setzt man für $c \in \mathbb{R}$ die beiden Indexmengen

$$I_0 = \{i \in \{1, \ldots, |V|\} \mid (e_2)_i \leq c\}$$
$$I_1 = \{i \in \{1, \ldots, |V|\} \mid (e_2)_i > c\}$$

und die Partitionsabbildung

$$\pi(v) = \begin{cases} 0 & \text{falls } v = v_i \wedge i \in I_0 \\ 1 & \text{falls } v = v_i \wedge i \in I_1 \end{cases}$$

Dabei wählt man das c so, dass die Arbeit gleichverteilt ist.

Kerninghan-Lin-Algorithmus

- Iterationsverfahren, das eine vorgegebene Partition unter Berücksichtigung des Kostenfunktionals für die Kommunikation (Kantengewichte des Graphen) verbessert.
- Das Verfahren von Kerningham-Lin wird meist in Kombination mit anderen Verfahren zur nachträglichen Verbesserung benutzt.
- Wir beschränken uns auf Bisektion (k-Wege Erweiterung möglich), es existieren also zwei Knotenmengen V_0 und V_1 mit $V_0 \cup V_1 = V$, die durch den Algorithmus in zwei neue Mengen $\bar{V_0}$ und $\bar{V_1}$ mit ebenfalls $\bar{V_0} \cup \bar{V_1} = V$ aber geringeren Kommunikationskosten überführt werden.
- Wir nehmen an, dass die Knotengewichte (die den Rechenaufwand beschreiben) eins sind, ein Tausch also die Lastverteilung nicht ändert.
- Außerdem sei die Anzahl der Knoten gerade.
- Grundidee des Verfahrens ist die Reduktion der Kosten durch Vertauschung von jeweils zwei Knoten zwischen den Partitionen.
- Sei $v_i \in V_j$ ein Knoten des Graphen, $E(v_i)$ sei die Summe der Kosten aller von v_i ausgehenden (externen) Kanten mit (v_i, v_l) mit $v_l \in V_{k \neq j}$ und $I(v_j)$ die Summe der Kosten aller von v_i ausgehenden internen Kanten (v_i, v_l) mit $v_l \in V_j$.
- Die Gesamtkosten sind durch die Summe der externen Kosten gegeben.
- Die Reduktion der Gesamtkosten durch einen Austausch von $v_i \in V_0$ und $v_j \in V_1$ ist dann

  $$[E(v_i) + E(v_j) - 2w(v_i, v_j)] - [I(v_i) + I(v_j)]\,,$$

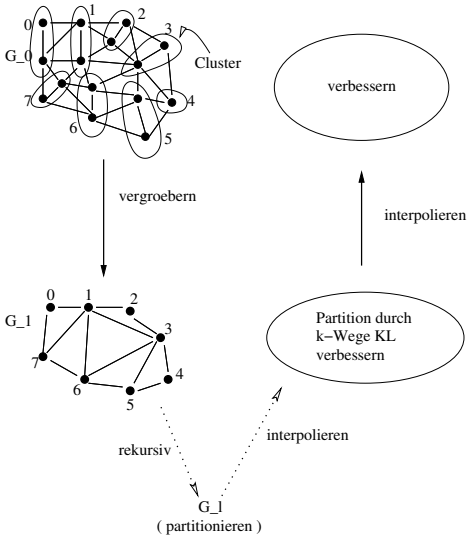
  da die bisherigen externen Kosten interne Kosten werden (bis auf die Kosten der Kante $w(v_i, v_j)$ zwischen v_i und v_j) und die bisherigen internen Kosten externe Kosten werden.

```
// Erzeuge initiale Partitionierung so, dass Gleichverteilung erfüllt ist.
while (1){ // Iterationsschritt
    V0' = V1' = ∅; // enthält später die schon getauschten Knoten
    V̄0 = V0, V̄1 = V1;
    for (k = 1 ; k ≤ |V|/2 ; k++)
    {
        // Wähle vi ∈ V0 \ V0' und vvi ∈ V1 \ V1' so,
        // dass g[k] = [E(vi) + E(vvi) − 2w(vi, vvi)] − [I(vi) + I(vvi)] maximal ist
        // setze
        V0' = V̄0' ∪ {vi},   V1' = V̄1' ∪ {vvi};
        V̄0 = (V̄0 \ {vi}) ∪ {vvi},   V̄1 = (V̄1 \ {vvi}) ∪ {vi};
    // berechne Kosten für alle mit vi und vj verbundenen Knoten neu
    } // for
    // Bem.: das Maximum darf negativ sein, d.h. Verschlechterung der Separatorkosten
    // Ergebnis an dieser Stelle: Folge von Paaren {(v1, vv1), ..., (vn, vvn)}.
```

// V_0, V_1 wurden noch nicht verändert.
// Wähle nun Teilfolge bis m $\leq |V|/2$, die maximale Verbesserung der Kosten bewirkt
// („hill climbing “), d.h. $\sum_{k=1}^{m} g[k] = \max$
$V_0 = V_0 \setminus \{v_1, \ldots, v_m\} \cup \{vv_1, \ldots, vv_m\}$;
$V_1 = V_1 \setminus \{vv_1, \ldots, vv_m\} \cup \{v_1, \ldots, v_m\}$;
if (m == 0) break; // Ende
} // while

Multilevel k-Wege Partitionierung

1. Zusammenfassen von Knoten des Ausgangsgraphen G^0 (z.B. zufällig oder aufgrund schwerer Kantengewichte) in Clustern.
2. Diese Cluster definieren die Knoten in einem vergröberten Graphen G^1.
3. Rekursiv führt dieses Verfahren auf einen Graphen G^l.



1. G^l wird nun partitioniert (z.B. RSB/KL)
2. Anschließend wird die Partitionsfunktion auf dem feineren Graphen G^{l-1} interpoliert.
3. Diese interpolierte Partitionsfunktion kann jetzt wieder mittels KL rekursiv verbessert und anschließend auf dem nächst feineren Graphen interpoliert werden.
4. So verfährt man rekursiv bis zum Ausgangsgraphen.
5. Die Implementierung ist dabei in $\mathcal{O}(N)$ Schritten möglich. Das Verfahren liefert eine qualitativ hochwertige Partitionierung

Weitere Probleme

Weitere Probleme bei der Partitionierung sind

- **Dynamische Repartitionierung:** Der Graph soll mit möglichst wenig Umverteilung lokal abgeändert werden.
- **Constraint Partitionierung:** Aus anderen Algorithmenteilen sind zusätzliche Datenabhängigkeiten vorhanden.
- **Parallelisierung des Partitionsverfahrens:** Ist nötig bei großen Datenmengen. Dafür gibt es fertige Software, z.B. ParMetis, die jedoch nicht unter allen Umständen ideal funktioniert.

15 Algorithmen für vollbesetzte Matrizen

15.1 Datenaufteilung von Vektoren und Matrizen

15.1.1 Aufteilung von Vektoren

- Vektor $x \in \mathbb{R}^N$ entspricht einer geordneten Liste von Zahlen.
- Jedem Index i aus der Indexmenge $I = \{0, \ldots, N-1\}$ wird eine reelle Zahl x_i zugeordnet.
- Anstelle von $\mathbb{R}^N$ können wir $\mathbb{R}(I)$ schreiben, um die Abhängigkeit von der Indexmenge deutlich zu machen.
- Die natürliche (und effizienteste) Datenstruktur für einen Vektor ist das Feld.
- Da Felder in vielen Programmiersprachen mit dem Index 0 beginnen, tut dies auch unsere Indexmenge I.
- Eine Datenaufteilung entspricht nun einer Zerlegung der Indexmenge I in

  $$I = \bigcup_{p \in P} I_p, \text{ mit } p \neq q \Rightarrow I_p \cap I_q = \emptyset,$$

  mit der Prozessmenge P.
- Bei guter Lastverteilung sollten die Indexmengen I_p, $p \in P$, jeweils (nahezu) gleich viele Elemente enthalten.
- Prozess $p \in P$ speichert somit die Komponenten x_i, $i \in I_p$, des Vektors x.
- In jedem Prozess möchten wir wieder mit einer zusammenhängenden Indexmenge $\tilde{I}_p$ arbeiten, die bei 0 beginnt, d.h.

  $$\tilde{I}_p = \{0, \ldots, |I_p| - 1\}.$$

Die Abbildungen

$$\begin{aligned} p\colon &\quad I \to P \text{ bzw.} \\ \mu\colon &\quad I \to \mathbb{N} \end{aligned}$$

ordnen jedem Index $i \in I$ umkehrbar eindeutig einen Prozess $p(i) \in P$ und einen lokalen Index $\mu(i) \in \tilde{I}_{p(i)}$ zu:

$$I \ni i \mapsto (p(i), \mu(i)).$$

Die Umkehrabbildung

$$\mu^{-1}\colon \underbrace{\bigcup_{p \in P} \{p\} \times \tilde{I}_p}_{\subset P \times \mathbb{N}} \to I$$

liefert zu jedem lokalen Index $i \in \tilde{I}_p$ und Prozess $p \in P$ den globalen Index $\mu^{-1}(p, i)$, d.h.

$$p(\mu^{-1}(p,i)) = p \text{ und } \mu(\mu^{-1}(p,i)) = i.$$

Gebräuchliche Aufteilungen sind vor allem die *zyklische Aufteilung* mit[5]

$$\begin{aligned} p(i) &= i \,\%\, P \\ \mu(i) &= i \div P \end{aligned}$$

und die *blockweise Aufteilung* mit

$$\begin{aligned} p(i) &= \begin{cases} i \div (B+1) & \text{falls } i < R \cdot (B+1) \\ R + [i - R \cdot (B+1)] \div B & \text{sonst} \end{cases} \\ \mu(i) &= \begin{cases} i \,\%\, (B+1) & \text{falls } i < R \cdot (B+1) \\ [i - R \cdot (B+1)] \,\%\, B & \text{sonst} \end{cases} \end{aligned}$$

mit $B = N \div P$ und $R = N \,\%\, P$. Hier ist die Idee, dass die ersten R Prozesse $B+1$ Indizes bekommen und die restlichen je B Indizes.

Zyklische und blockweise Aufteilung für $N = 13$ und $P = 4$:

zyklische Aufteilung:

I:	0	1	2	3	4	5	6	7	8	9	10	11	12
$p(i)$:	0	1	2	3	0	1	2	3	0	1	2	3	0
$\mu(i)$:	0	0	0	0	1	1	1	1	2	2	2	2	3

z.B. $I_1 = \{1, 5, 9\}$,
$\tilde{I}_1 = \{0, 1, 2\}$.

blockweise Aufteilung

I:	0	1	2	3	4	5	6	7	8	9	10	11	12
$p(i)$:	0	0	0	0	1	1	1	2	2	2	3	3	3
$\mu(i)$:	0	1	2	3	0	1	2	0	1	2	0	1	2

z.B. $I_1 = \{4, 5, 6\}$,
$\tilde{I}_1 = \{0, 1, 2\}$.

15.1.2 Aufteilung von Matrizen

- Bei einer Matrix $A \in \mathbb{R}^{N \times M}$ wird jedem Tupel $(i, j) \in I \times J$, mit $I = \{0, \ldots, N-1\}$ und $J = \{0, \ldots, M-1\}$ eine reele Zahl a_{ij} zugeordnet.
- Prinzipiell beliebige Zuordnung von Matrixelementen zu Prozessoren
- Jedoch können die einem Prozessor zugeordneten Elemente im allgemeinen *nicht* wieder als Matrix dargestellt werden.
- Ausnahme: separate Zerlegung der eindimensionalen Indexmengen I und J.
- Dazu nehmen wir an, die Prozesse seien als zweidimensionales Feld organisiert, d.h.

  $$(p, q) \in \{0, \ldots, P-1\} \times \{0, \ldots, Q-1\}.$$

- Die Indexmengen I, J werden zerlegt in

  $$I = \bigcup_{p=0}^{P-1} I_p \text{ und } J = \bigcup_{q=0}^{Q-1} I_q$$

[5] $\div$ bedeutet ganzzahlige Division; % die modulo-Funktion

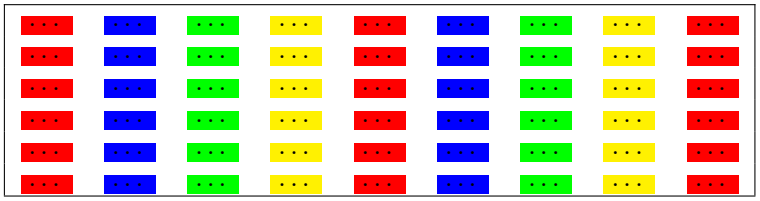
- Prozess (p, q) ist dann für die Indizes $I_p \times I_q$ verantwortlich.
- Lokal speichert Prozess (p, q) seine Elemente dann als $\mathbb{R}(\tilde{I}_p \times \tilde{J}_q)$-Matrix.
- Die Zerlegungen von I und J werden formal durch die Abbildungen p und μ sowie q und ν beschrieben:

$$I_p = \{i \in I \mid p(i) = p\}, \quad \tilde{I}_p = \{n \in \mathbb{N} \mid \exists i \in I : p(i) = p \wedge \mu(i) = n\}$$
$$J_q = \{j \in J \mid q(j) = q\}, \quad \tilde{J}_q = \{m \in \mathbb{N} \mid \exists j \in J : q(j) = q \wedge \nu(j) = m\}$$

Beispiele zur Aufteilung einer 6×9-Matrix auf vier Prozessoren

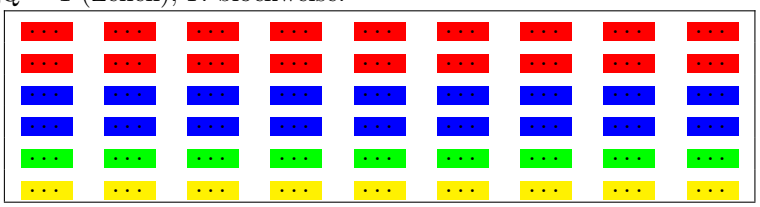
(a) $P = 1$, $Q = 4$ (Spalten), J: zyklisch:

0	1	2	3	4	5	6	7	8	J
0	1	2	3	0	1	2	3	0	q
0	0	0	0	1	1	1	1	2	ν



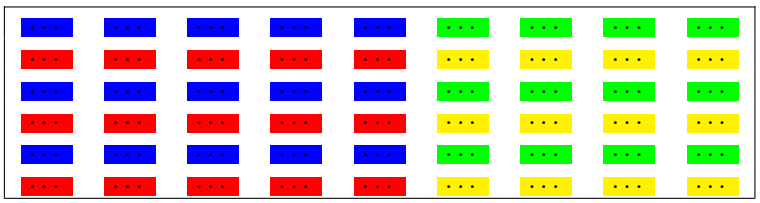
(b) $P = 4$,$Q = 1$ (Zeilen), I: blockweise:

0	0	0
1	0	1
2	1	0
3	1	1
4	2	0
5	3	0
I	p	μ



(c) $P = 2$, $Q = 2$ (Feld), I: zyklisch, J: blockweise:

			0	1	2	3	4	5	6	7	8	J
			0	0	0	0	0	1	1	1	1	q
			0	1	2	3	4	0	1	2	3	ν
0	0	0										
1	1	0										
2	0	1										
3	1	1										
4	0	2										
5	1	2										
I	p	μ										



Welche Datenaufteilung ist nun die Beste?

- Generell liefert die Organisation der Prozesse als möglichst quadratisches Feld eine Aufteilung mit guter Lastverteilung.
- Wichtiger ist jedoch, dass sich unterschiedliche Aufteilungen unterschiedlich gut für verschiedene Algorithmen eignen.

- Wir werden sehen, dass sich ein Prozessfeld mit zyklischer Aufteilung sowohl der Zeilen aus auch der Spaltenindizes recht gut für die LU-Zerlegung eignet.

- Diese Aufteilung ist jedoch nicht optimal für die Auflösung der entstehenden Dreiecks-systeme. Muss man das Gleichungssystem für viele rechte Seiten lösen, so ist ein Kompromiss anzustreben.

- Dies gilt generell für fast alle Aufgaben aus der linearen Algebra: Die Multiplikation zweier Matrizen oder die Transposition einer Matrix stellt nur einen Schritt eines größeren Algorithmus dar.

- Die Datenaufteilung kann somit nicht auf einen Teilschritt hin optimiert werden, sondern sollte einen guten Kompromiss darstellen. Eventuell kann auch überlegt werden, ob ein Umkopieren der Daten sinnvoll ist.

15.2 Transponieren einer Matrix

Aufgabenstellung Gegeben: $A \in \mathbb{R}^{N \times M}$, verteilt auf eine Menge von Prozessen; Bestimme: A^T mit der selben Datenaufteilung wie A.

- Im Prinzip ist das Problem trivial.

- Wir könnten die Matrix so auf die Prozessoren aufteilen, dass nur Kommunikation mit nächsten Nachbarn notwendig ist (da die Prozesse paarweise kommunizieren).

12	1	3	5
0	13	7	9
2	6	14	11
4	8	10	15

Optimale Datenaufteilung für die Matrixtransposition (die Zahlen bezeichnen die Prozessornumern).

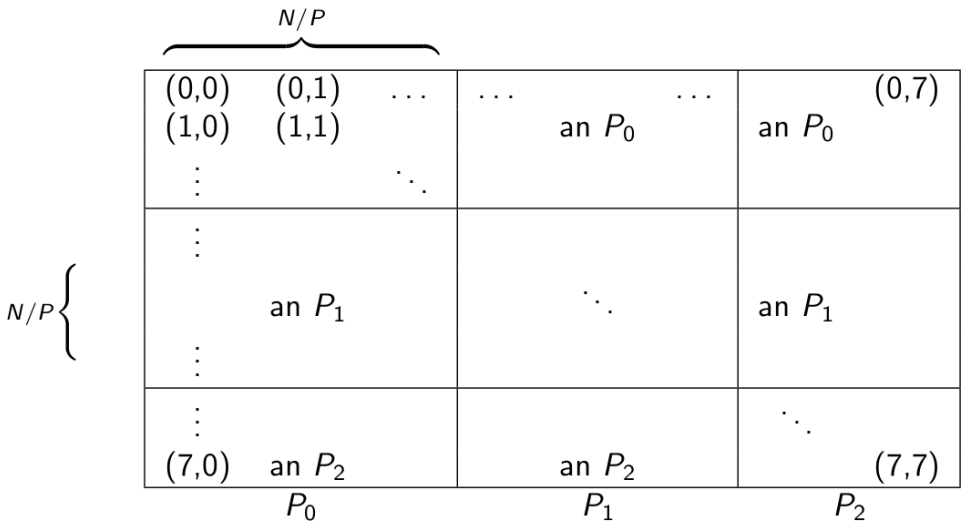
Beispiel mit Ringtopologie:

- Offensichtlich ist nur Kommunikation zwischen direkten Nachbarn notwendig ($0 \leftrightarrow 1, 2 \leftrightarrow 3, \ldots, 10 \leftrightarrow 11$).

- Allerdings entspricht diese Datenaufteilung nicht dem Schema, das wir eben eingeführt haben und eignet sich z.B. weniger gut für die Multiplikation zweier Matrizen.

1D Aufteilung

Betrachten wir o.B.d.A eine spaltenweise, geblockte Aufteilung



8×8-Matrix auf drei Prozessoren in spaltenweiser, geblockter Aufteilung.

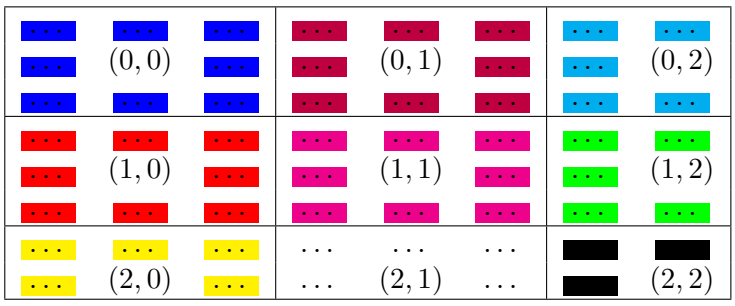
- Offensichtlich hat in diesem Fall jeder Prozessor Daten an jeden anderen zu senden.
- Es handelt sich also um all-to-all mit persönlichen Nachrichten.
- Nehmen wir eine Hypercubestruktur als Verbindungstopologie an, so erhalten wir folgende parallele Laufzeit für eine $N \times N$-Matrix und P Prozessoren:

$$\begin{aligned} T_P(N,P) &= \underbrace{2(t_s+t_h)\operatorname{ld} P}_{\text{Aufsetzen}} + \underbrace{t_w \frac{N^2}{P^2} P \operatorname{ld} P}_{\text{Datenübertragung}} + \underbrace{(P-1)\frac{N^2}{P^2}\frac{t_e}{2}}_{\text{Transponieren}} \\ &\approx 2\operatorname{ld} P \cdot (t_s+t_h) + \frac{N^2}{P}\operatorname{ld} P \cdot t_w + \frac{N^2}{P}\frac{t_e}{2} \end{aligned}$$

- Selbst bei festem P und wachsendem N können wir den Anteil der Kommunikation an der Gesamtlaufzeit nicht beliebig klein machen.
- Dies ist bei allen Algorithmen zur Transposition so (auch bei der optimalen Aufteilung oben).
- Matrixtransposition besitzt also keine Isoeffizienzfunktion und ist somit nicht skalierbar.

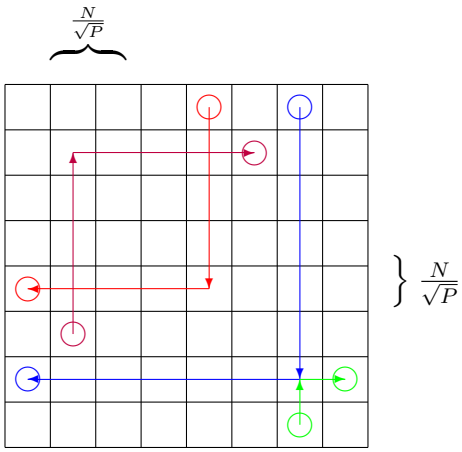
2D Aufteilung

Wir betrachten nun eine zweidimensionale, geblockte Aufteilung einer $N \times N$-Matrix auf ein $\sqrt{P} \times \sqrt{P}$-Prozessorfeld:



Beispiel für eine zweidimensionale, geblockte Aufteilung $N = 8$, $\sqrt{P} = 3$.

- Jeder Prozessor muss seine Teilmatrix mit genau einem anderen austauschen.
- Ein naiver Transpositionsalgorithmus für diese Konfiguration ist:
  - Prozessoren (p, q) unterhalb der Hauptdiagonalen $(p > q)$ schicken Teilmatrix in der Spalte nach oben bis zu Prozessor (q, q), danach läuft die Teilmatrix nach rechts bis in die richtige Spalte zu Prozessor (q, p).
  - Entsprechend laufen Daten von Prozessoren (p, q) oberhalb der Hauptdiagonalen $(q > p)$ erst in der Spalte q nach unten bis (q, q) und dann nach links bis (q, p) erreicht wird.



Diverse Wege von Teilmatrizen bei $\sqrt{P} = 8$.

- Offensichtlich leiten Prozessoren (p, q) mit $p > q$ Daten von unten nach oben bzw. rechts nach links und Prozessoren (p, q) mit $q > p$ entsprechend Daten von oben nach unten und links nach rechts.
- Bei synchroner Kommunikation sind in jedem Schritt vier Sende- bzw. Empfangsoperationen notwendig, und insgesamt braucht man $2(\sqrt{P} - 1)$ Schritte.
- Die parallele Laufzeit beträgt somit

$$\begin{aligned} T_P(N,P) &= 2(\sqrt{P}-1)\cdot 4\left(t_s + t_h + t_w\left(\frac{N}{\sqrt{P}}\right)^2\right) + \frac{1}{2}\left(\frac{N}{\sqrt{P}}\right)^2 t_e \\ &\approx 8\sqrt{P}\cdot(t_s + t_h) + 8\frac{N^2}{P}\sqrt{P}\cdot t_w + \frac{N^2}{P}\cdot\frac{t_e}{2} \end{aligned}$$

- Im Vergleich zur eindimensionalen Aufteilung mit Hypercube hat man in der Datenübertragung den Faktor $\sqrt{P}$ statt $\operatorname{ld} P$.

Rekursiver Transpositionsalgorithmus

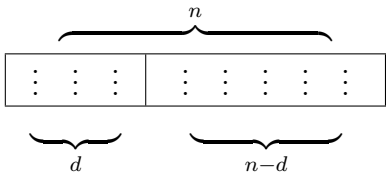
Dieser Algorithmus basiert auf folgender Beobachtung: Für eine 2×2-Blockmatrixzerlegung von A gilt

$$A^T = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix}^T = \begin{pmatrix} A_{00}^T & A_{10}^T \\ A_{01}^T & A_{11}^T \end{pmatrix}$$

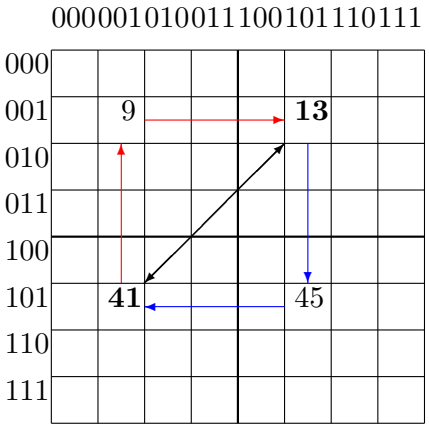
d.h. die Nebendiagonalblöcke tauschen die Plätze und dann muss jede Teilmatrix transponiert werden. Dies geschieht natürlich rekursiv bis eine 1×1-Matrix erreicht ist. Ist $N = 2^n$, so sind n Rekursionsschritte notwendig.

- Der Hypercube ist die ideale Verbindungstopologie für diesen Algorithmus.
- Mit $N = 2^n$ und $\sqrt{P} = 2^d$ mit $n \geq d$ geschieht die Zuordnung der Indizes $I = \{0, \ldots, N-1\}$ auf die Prozessoren mittels

$$\begin{aligned} p(i) &= i \div 2^{n-d}, \\ \mu(i) &= i \,\%\, 2^{n-d} \end{aligned}$$

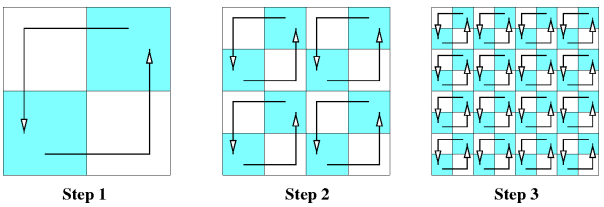


- Die oberen d Bits eines Index beschreiben den Prozessor, auf den der Index abgebildet wird. Analog für q.
- Betrachten wir als Beispiel $d = 3$, d.h. $\sqrt{P} = 2^3 = 8$.



Kommunikation im ersten Schritt des rekursiven Transpositionsalgorithmus bei $d = 3$.

- Im ersten Rekursionsschritt ist die Matrix in 2×2 Blöcke aus 4×4-Teilmatrizen zu teilen und $4 \cdot 16$ Prozessoren müssen Daten austauschen, z.B. Prozessor $101001 = 41$ und $001101 = 13$. Dies geschieht in zwei Schritten über die Prozessoren $001001 = 9$ und $101101 = 45$.
- Diese sind beide *direkte* Nachbarn der Prozessoren 41 und 13 im Hypercube.



- Der rekursive Transpositionsalgorithmus arbeitet nun rekursiv über die Prozessortopologie.

- Im nächsten Schritt werden 4×4 Blöcke aus 2×2 Teilmatrizen zwischen 16×4 Prozessen ausgetauscht.
- Im dritten Schritt werden 8×8 Blöcke aus 1×1 Teilmatrizen zwischen 64×1 Prozessen ausgetauscht. Danach ist *ein* Prozessor erreicht, und die Blöcke werden mit dem sequentiellen Algorithmus transponiert.

Die parallele Laufzeit beträgt somit

$$T_P(N,P) = 2 \operatorname{ld} \sqrt{P} \cdot (t_s + t_h) + 2\frac{N^2}{P} \operatorname{ld} \sqrt{P} \cdot t_w + \frac{N^2}{P}\frac{t_e}{2}$$

Programm 15.1 (Rekursiver Transpositionsalgorithmus auf Hypercube).

```
parallel recursive-transpose
{
    const int d = ..., n = ...;
    const int P = 2^d, N = 2^n;

    process Π[int (p,q) ∈ {0,...,2^d − 1} × {0,...,2^d − 1}]
    {
        Matrix A, B;                    // A ist die Eingabematrix
        void rta(int r, int s, int k)
        {
            if (k == 0) { A = A^T; return; }
            int i = p − r, j = q − s, l = 2^(k−1);
            if (i < l)
            {
                if (j < l)
                {                       // links oben
                    recv(B,Π_{p+l,q}); send(B,Π_{p,q+l});
                    rta(r,s,k − 1);
                }
                else
                {                       // rechts oben
                    send(A,Π_{p+l,q}); recv(A,Π_{p,q−l});
                    rta(r,s + l,k − 1);
                }
            }
            ...
}
```

Programm 15.2 (Rekursiver Transpositionsalgorithmus auf Hypercube cont.).

```
parallel recursive-transpose cont.
{

            ...
            else
```

```
        {
            if (j < l) {              // links unten
                send(A,Π_{p-l,q}); recv(A,Π_{p,q+l});
                rta(r + l,s,k − 1);
            }
            else
            {                         // rechts unten
                recv(B,Π_{p-l,q}); send(B,Π_{p,q-l});
                rta(r + l,s + l,k − 1);
            }
        }
    }
    rta(0,0,d);
  }
}
```

15.3 Matrix-Vektor Multiplikation

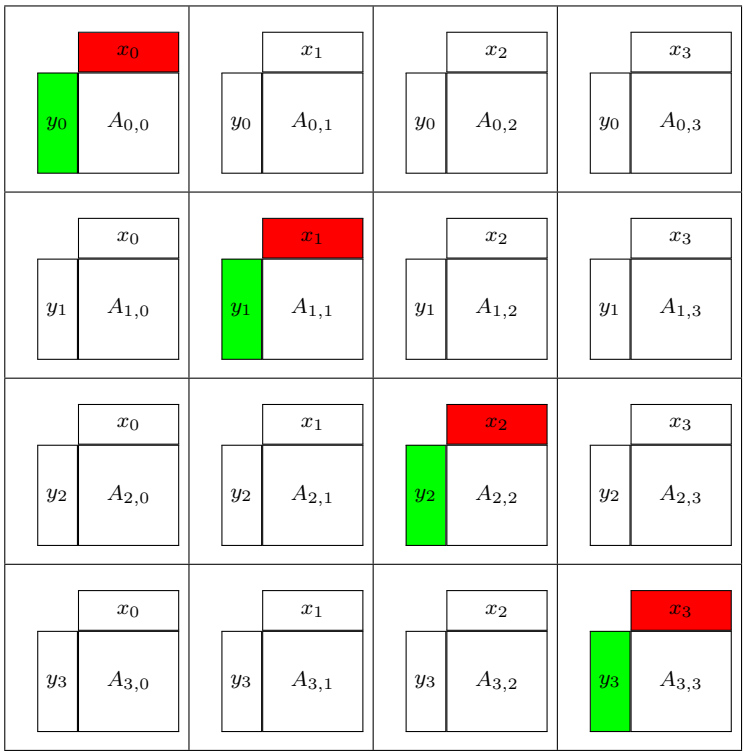
Berechne $y = Ax$, Matrix $A \in \mathbb{R}^{N \times M}$ und Vektor $x \in \mathbb{R}^M$

- Unterschiedliche Möglichkeiten zur Datenaufteilung
- Verteilung der Matrix und des Vektors müssen aufeinander abgestimmt sein
- Verteilung des Ergebnisvektor $y \in \mathbb{R}^N$ wie bei Eingabevektor x

Beispiel:

- Matrix sei blockweise auf eine Feldtopologie verteilt
- Eingabevektor x entsprechend blockweise über die Diagonalprozessoren verteilt
- das Prozessorfeld ist quadratisch
- Vektorsegment x_q wird in jeder Prozessorspalte benötigt und ist somit in jeder Spalte zu kopieren (einer-an-alle).
- lokale Berechnung des Produkts $y_{p,q} = A_{p,q}x_q$.
- komplettes Segment y_p ergibt sich erst duch die Summation $y_p = \sum_q y_{p,q}$. (weitere alle-an-einen Kommunikation)
- Resultat kann unmittelbar für weitere Matrix-Vektor-Multiplikation benutzt werden

Aufteilung für das Matrix-Vektor-Produkt



Parallele Laufzeit für eine $N \times N$-Matrix und $\sqrt{P} \times \sqrt{P}$ Prozessoren mit cut-through Kommunikationsnetzwerk:

$$
\begin{aligned}
T_P(N,P) &= \underbrace{\left(t_s + t_h + t_w \overbrace{\frac{N}{\sqrt{P}}}^{\text{Vektor}}\right) \operatorname{ld}\sqrt{P}}_{\substack{\text{Austeilen von} \\ x \text{ über Spalte}}} + \underbrace{\left(\frac{N}{\sqrt{P}}\right)^2 2t_f}_{\substack{\text{lokale Matrix-} \\ \text{Vektor-Mult.}}} \\
&\quad + \underbrace{\left(t_s + t_h + t_w \frac{N}{\sqrt{P}}\right) \operatorname{ld}\sqrt{P}}_{\substack{\text{Reduktion} \\ (t_f \ll t_w)}} = \\
&= \operatorname{ld}\sqrt{P}(t_s + t_h)2 + \frac{N}{\sqrt{P}} \operatorname{ld}\sqrt{P} 2t_w + \frac{N^2}{P} 2t_f
\end{aligned}
$$

Für festes P und $N \to \infty$ wird der Kommunikationsanteil beliebig klein, es existiert also eine Isoeffizienzfunktion, der Algorithmus ist skalierbar.

Berechnen wir Arbeit und Overhead:

Umrechnen auf die Arbeit W:

$$
\begin{aligned}
W &= N^2 2t_f \text{ (seq. Laufzeit)} \\
&\Rightarrow N = \frac{\sqrt{W}}{\sqrt{2t_f}} \\
T_P(W,P) &= \operatorname{ld}\sqrt{P}(t_s + t_h)2 + \frac{\sqrt{W}}{\sqrt{P}} \operatorname{ld}\sqrt{P} \frac{2t_w}{\sqrt{2t_f}} + \frac{W}{P}
\end{aligned}
$$

Overhead:

$$T_O(W,P) = PT_P(W,P) - W =$$
$$= \sqrt{W}\sqrt{P}\,\mathrm{ld}\,\sqrt{P}\frac{2t_w}{\sqrt{2t_f}} + P\,\mathrm{ld}\,\sqrt{P}(t_s+t_h)2$$

und nun die Isoeffizienzfunktion:

Isoeffizienz ($T_O(W,P) \stackrel{!}{=} KW$): T_O hat zwei Terme. Für den ersten erhalten wir

$$\sqrt{W}\sqrt{P}\,\mathrm{ld}\,\sqrt{P}\frac{2t_w}{\sqrt{2t_f}} = KW$$
$$\iff \quad W = P(\mathrm{ld}\,\sqrt{P})^2\frac{4t_w^2}{2t_fK^2}$$

und für den zweiten

$$P\,\mathrm{ld}\,\sqrt{P}(t_s+t_h)2 = KW$$
$$\iff \quad W = P\,\mathrm{ld}\,\sqrt{P}\frac{(t_s+t_h)2}{K};$$

somit ist $W = \Theta(P(\mathrm{ld}\,\sqrt{P})^2)$ die gesuchte Isoeffizienzfunktion. Wegen $N = \sqrt{\frac{W}{2t_f}}$ heißt das, dass die Blockgröße $\frac{N}{\sqrt{P}}$ wie $\mathrm{ld}\,\sqrt{P}$ wachsen muss.

15.4 Matrix-Matrix Multiplikation

15.4.1 Algorithmus von Cannon

Es ist $C = A \cdot B$ zu berechnen.

- Zu multiplizierende $N \times N$-Matrizen A und B sind blockweise auf eine 2D-Feldtopologie ($\sqrt{P} \times \sqrt{P}$) verteilt
- Praktischerweise soll das Ergebnis C wieder in derselben Verteilung vorliegen.
- Prozess (p,q) muss somit

  $$C_{p,q} = \sum_k A_{p,k} \cdot B_{k,q}$$

  berechnen, benötigt also Blockzeile p von A und Blockspalte q von B.

Die zwei Phasen des Algorithmus von Cannon:

1. *Alignment-Phase:* Die Blöcke von A werden in jeder Zeile zyklisch nach links geschoben, bis der Diagonalblock in der ersten Spalte zu liegen kommt. Entsprechend schiebt man die Blöcke von B in den Spalten nach oben, bis alle Diagonalblöcke in der ersten Zeile liegen.

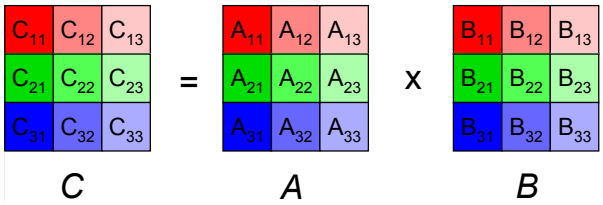
   Nach der Alignment-Phase hat Prozessor (p,q) die Blöcke

   $A_{p,\underbrace{(q+p)\,\%\,\sqrt{P}}}$ (Zeile p schiebt p mal nach links)

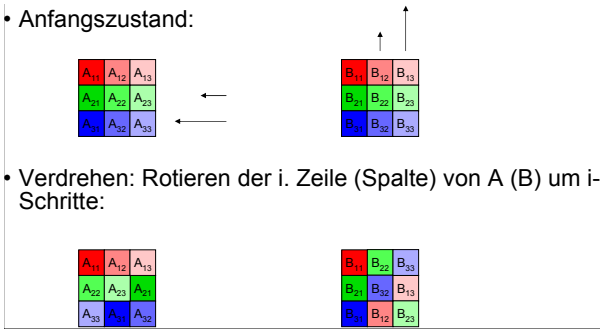
   $B_{\underbrace{(p+q)\,\%\,\sqrt{P}},q}$ (Spalte q schiebt q mal nach oben).

2. *Rechenphase:* Offensichtlich verfügt nun jeder Prozess über zwei passende Blöcke, die er multiplizieren kann. Schiebt man die Blöcke von A in jeder Zeile von A zyklisch um eine Position nach links und die von B in jeder Spalte nach oben, so erhält jeder wieder zwei passende Blöcke. Nach $\sqrt{P}$ Schritten ist man fertig.

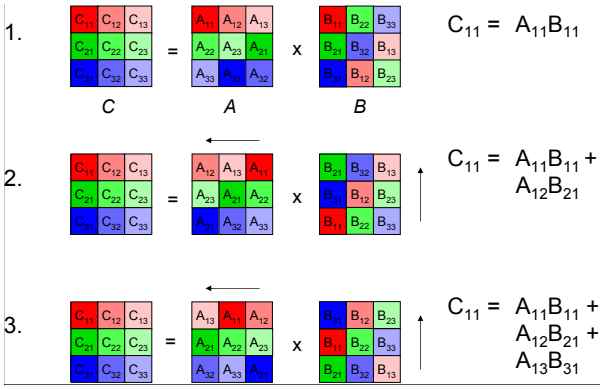
- basiert auf blockweise Partitionierung der Matrizen
- Setup-Phase
  - Rotation der Matrizen A und B
- Iteration über $\sqrt{p}$
  - Berechne lokales Block-Matrix-Produkt
  - Shift A horizontal und B vertikal



Rotation



Iteration



Cannon mit MPI (Init)

```
// Baue Gitter und hole Koordinaten
int dims[2];
int periods[2] = {1,1};
int mycoords[2];
MPI_Comm comm2d;

dims[0] = dims[1] = (int)sqrt(numProcs);

MPI_Cart_create(MPI_COMM_WORLD,2,dims,periods,0,&comm2d);
MPI_Comm_rank(comm2d,&my2drank);
MPI_Cart_coords(comm2d,my2drank,2,mycoords);

// Lokale Bloecke der Matrizen
double *A, *B, *C;

// Lade A, B entsprechend der Koordinaten
// Allokiere und initialisiere C
```

Cannon mit MPI (Rotate)

```
// Matrix-Verdrehung A
MPI_Cart_shift(comm2d,0,-mycoord[1],&shiftsource,&shiftdest);
MPI_Sendrecv_replace(A,nlocal*nlocal,MPI_DOUBLE,
                     shiftdest,0,shiftsource,0,comm2d,&status);

// Matrix-Verdrehung B
MPI_Cart_shift(comm2d,0,-mycoord[1],&shiftsource,&shiftdest);
MPI_Sendrecv_replace(B,nlocal*nlocal,MPI_DOUBLE,
                     shiftdest,0,shiftsource,0,comm2d,&status);
```

Cannon mit MPI (Iteration)

```
// Berechnung der Partner fuer Matrix-Verdrehung von A und B
MPI_Cart_shift(comm2d,1,-1,&rightrank,&leftrank);
MPI_Cart_shift(comm2d,0,-1,&downrank,&uprank);

for (int i=0;i<dims[0];++i)
{
  dgemm(nlocal,A,B,C); // C = C + A * B

  // Matrix A nach links rollen
  MPI_Sendrecv_replace(A,nlocal*nlocal,MPI_DOUBLE,
                       leftrank,0,rightrank,0,comm2d,&status);

  // Matrix B nach oben rollen
```

```
    MPI_Sendrecv_replace(B,nlocal*nlocal,MPI_DOUBLE,
15                         uprank,0,downrank,0,comm2d,&status);
  }
17
  // versetzte A und B zurueck in Ursprungs-Zustand
19 // wie Rotate nur mit umgekehrten Vorzeichen der Verschiebung
```

Performance Analysis

- A, B rotieren: $4 \cdot \left(\sqrt{P} - 1\right) \cdot \left(t_s + t_h + t_w \cdot \frac{N^2}{P}\right)$
- Pro Iteration:
  - Lokale Matrixmultiplikation: $2 \cdot t_f \cdot \left(\frac{N}{\sqrt{P}}\right)^3 = 2 \cdot t_f \cdot \frac{N^3}{P^{\frac{3}{2}}}$
  - A, B einmal rotieren: $4 \cdot \left(t_s + t_h + t_w \cdot \frac{N^2}{P}\right)$ außer im letzten Schritt
- Gesamt: $t_{\mathsf{cannon}}(P) = 8 \cdot (t_s + t_h) \cdot \left(\sqrt{P} - 1\right) + 8 \cdot t_w \cdot \frac{N^2}{\sqrt{P}} + 2 \cdot t_f \cdot \frac{N^3}{P}$
- Effizienz:

$$\begin{aligned} E &= \frac{2 \cdot t_f \cdot N^3}{P \cdot t_{\mathsf{cannon}}(P)} \\ &\approx \frac{1}{4 \cdot \frac{t_s + t_h}{t_f} \cdot \left(\frac{\sqrt{P}}{N}\right)^3 + 4 \cdot \frac{t_w}{t_f} \cdot \frac{\sqrt{P}}{N} + 1} \end{aligned}$$

- Effizienz $\to 1$, wenn $\frac{N}{\sqrt{P}} \to \infty$

Isoeffizienzanalyse

Betrachten wir die zugehörige Isoeffizienzfunktion.

Sequentielle Laufzeit :

$$\begin{aligned} W = T_S(N) &= N^3 2 t_f \\ \Rightarrow N &= \left(\frac{W}{2t_f}\right)^{\frac{1}{3}} \end{aligned}$$

parallele Laufzeit:

$$\begin{aligned} T_P(N,P) &\approx 8 \cdot (t_s + t_h) \cdot \sqrt{P} + 8 \cdot t_w \cdot \frac{N^2}{\sqrt{P}} + 2 \cdot t_f \cdot \frac{N^3}{P} \\ T_P(W,P) &= 8 \cdot (t_s + t_h) \cdot \sqrt{P} + \frac{W^{\frac{2}{3}}}{\sqrt{P}} \frac{8 t_w}{(2t_f)^{\frac{1}{3}}} + \frac{W}{P} \end{aligned}$$

Overhead:

$$T_O(W,P) = PT_P(W,P) - W = 8 \cdot (t_s + t_h) \cdot P^{\frac{3}{2}} + \sqrt{P}W^{\frac{2}{3}} \frac{8t_w}{(2t_f)^{\frac{1}{3}}}$$

Ergebnis:

- Somit ist $W = \Theta(P^{3/2})$.
- Wegen $N = \left(\frac{W}{2t_f}\right)^{1/3}$ ist die Effizienz konstant falls $N/\sqrt{P} = const$
- Somit ist bei *fester* Größe der Blöcke in jedem Prozessor und wachsender Prozessorzahl die Effizienz konstant.
- Beschränken wir uns beim Algorithmus von Cannon auf 1×1-Blöcke pro Prozessor, also $\sqrt{P} = N$, so können wir für die erforderlichen N^3 Multiplikationen nur N^2 Prozessoren nutzen.
- Dies ist der Grund für die Isoeffizienzfunktion der Ordnung $P^{3/2}$.

Praktische Aspekte

- effiziente, aber nicht einfache Verallgemeinerung, falls
  - Matrizen nicht quadratisch sind
  - Dimensionen sind nicht ohne Rest durch P teilbar
  - andere Matrix Paritionierungen gebraucht werden
- Dekel-Nassimi-Salmi-Algorithmus erlaubt die Nutzung von N^3 Prozessoren (Cannon N^2) mit besserer Isoeffizienz Funktion.

15.4.2 Dekel-Nassimi-Salmi-Algorithmus

- Nun betrachten wir einen Algorithmus der den Einsatz von bis zu N^3 Prozessoren bei einer $N \times N$-Matrix erlaubt.
- Gegeben seien also $N \times N$-Matrizen A und B sowie ein 3D-Feld von Prozessoren der Dimension $P^{1/3} \times P^{1/3} \times P^{1/3}$.
- Die Prozessoren werden über die Koordinaten (p, q, r) adressiert.
- Um den Block $C_{p,q}$ der Ergebnismatrix C mittels

$$C_{p,q} = \sum_{r=0}^{P^{\frac{1}{3}}-1} A_{p,r} \cdot B_{r,q} \tag{27}$$

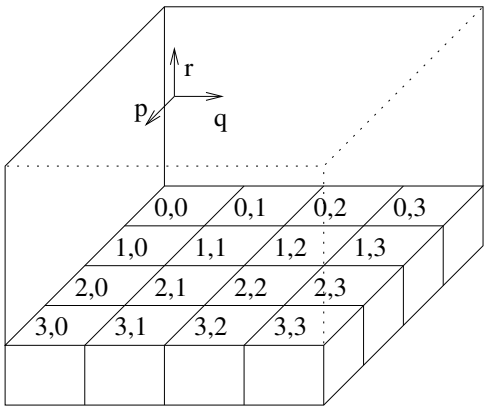
  zu berechnen, setzen wir $P^{1/3}$ Prozessoren ein, und zwar ist Prozessor (p, q, r) genau für das Produkt $A_{p,r} \cdot B_{r,q}$ zuständig.
- Nun ist noch zu klären, wie Eingabe- und Ergebnismatrizen verteilt sein sollen.

- Sowohl A als auch B sind in $P^{1/3} \times P^{1/3}$-Blöcke der Größe $\frac{N}{P^{1/3}} \times \frac{N}{P^{1/3}}$ zerlegt.

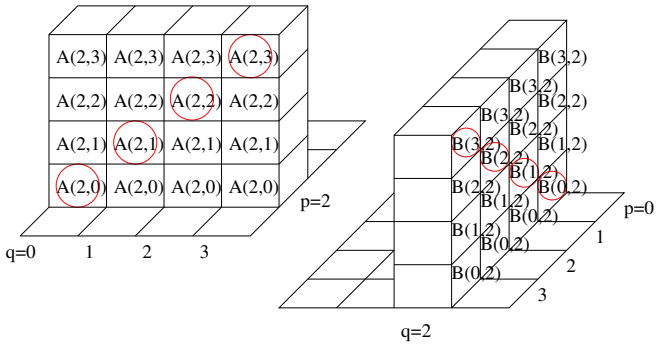
- $A_{p,q}$ und $B_{p,q}$ wird zu Beginn in Prozessor $(p, q, 0)$ gespeichert, auch das Ergebnis $C_{p,q}$ soll dort liegen.

- Die Prozessoren (p, q, r) für $r > 0$ werden nur zwischenzeitlich benutzt.

Verteilung von A, B, C für $P^{1/3} = 4$ (P=64).



Verteilung der Blöcke von A und B (zu Beginn) und C (am Ende)



Verteilung von A und B für die Multiplikation

- Damit nun jeder Prozessor (p, q, r) „seine“ Multiplikation $A_{p,r} \cdot B_{r,q}$ durchführen kann, sind die beteiligten Blöcke von A und B erst an ihre richtige Position zu befördern.

- Alle Prozessoren benötigen $(p, *, r)$ den Block $A_{p,r}$ und alle Prozessoren $(*, q, r)$ den Block $B_{r,q}$.

- Sie wird folgendermaßen hergestellt:

  Prozessor $(p, q, 0)$ sendet $A_{p,q}$ an Prozessor (p, q, q) und dann sendet (p, q, q) das $A_{p,q}$ an alle $(p, *, q)$ mittels einer einer-an-alle Kommunikation auf $P^{1/3}$ Prozessoren. Entsprechend schickt $(p, q, 0)$ das $B_{p,q}$ an Prozessor (p, q, p), und dieser verteilt dann an $(*, q, p)$.

- Nach der Multiplikation in jedem (p, q, r) sind die Ergebnisse aller $(p, q, *)$ noch in $(p, q, 0)$ mittels einer alle-an-einen Kommunikation auf $P^{1/3}$ Prozessoren zu sammeln.

Analysieren wir das Verfahren im Detail (3D-cut-through Netzwerk):

$$
\begin{aligned}
W &= T_S(N) = N^3 2t_f \quad \Rightarrow N = \left(\frac{W}{2t_f}\right)^{\frac{1}{3}} \\
T_P(N,P) &= \underbrace{\left(\underbrace{t_s + t_h + t_w\left(\frac{N}{P^{\frac{1}{3}}}\right)^2}_{(p,q,0)\longrightarrow(p,q,q),(p,q,p)}\right)}^{A_{p,q}\text{ u. }B_{p,q}}\overbrace{2} + \underbrace{\left(t_s + t_h + t_w\left(\frac{N}{P^{\frac{1}{3}}}\right)^2\right)\operatorname{ld} P^{\frac{1}{3}}}_{\text{einer-an-alle}}\overbrace{2}^{A,B} \\
&\quad + \underbrace{\left(\frac{N}{P^{\frac{1}{3}}}\right)^3 2t_f}_{\text{Multiplikation}} + \underbrace{\left(t_s + t_h + t_w\left(\frac{N}{P^{\frac{1}{3}}}\right)^2\right)\operatorname{ld} P^{\frac{1}{3}}}_{\substack{\text{alle-an-einen}\\(t_f \ll t_w)}} \approx \\
&\approx 3\operatorname{ld} P^{\frac{1}{3}}(t_s + t_h) + \frac{N^2}{P^{\frac{2}{3}}} 3\operatorname{ld} P^{\frac{1}{3}} t_w + \frac{N^3}{P} 2t_f \\
T_P(W,P) &= 3\operatorname{ld} P^{\frac{1}{3}}(t_s + t_h) + \frac{W^{\frac{2}{3}}}{P^{\frac{2}{3}}} 3\operatorname{ld} P^{\frac{1}{3}} \frac{t_w}{(2t_f)^{\frac{2}{3}}} + \frac{W}{P} \\
T_O(W,P) &= P\operatorname{ld} P^{\frac{1}{3}} 3(t_s + t_h) + W^{\frac{2}{3}} P^{\frac{1}{3}} \operatorname{ld} P^{\frac{1}{3}} \frac{3t_w}{(2t_f)^{\frac{2}{3}}}
\end{aligned}
$$

- Aus dem zweiten Term von $T_O(W,P)$ nähern wir die Isoeffizienzfunktion an:

$$
\begin{aligned}
& W^{\frac{2}{3}} P^{\frac{1}{3}} \operatorname{ld} P^{\frac{1}{3}} \frac{3t_w}{(2t_f)^{\frac{2}{3}}} = KW \\
\iff\quad & W^{\frac{1}{3}} = P^{\frac{1}{3}} \operatorname{ld} P^{\frac{1}{3}} \frac{3t_w}{(2t_f)^{\frac{2}{3}} K} \\
\iff\quad & \boxed{W = P\left(\operatorname{ld} P^{\frac{1}{3}}\right)^3 \frac{27{t_w}^3}{4{t_f}^2 K^3}.}
\end{aligned}
$$

- Also erhalten wir die Isoeffizienzfunktion $O(P \cdot (\operatorname{ld} P)^3)$ und somit eine bessere Skalierbarkeit als für den Cannon'schen Algorithmus.
- Wir haben immer angenommen, dass die optimale sequentielle Komplexität der Matrixmultiplikation N^3 ist. Der Algorithmus von Strassen hat jedoch eine Komplexität von $O(N^{2.87})$.
- Für eine effiziente Implementierung der Multiplikation zweier Matrixblöcke auf einem Prozessor muß auf Cacheeffizienz geachtet werden.

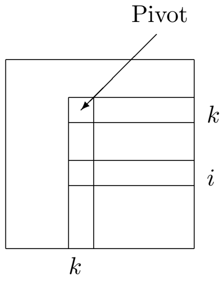
15.5 LU-Zerlegung

Zu lösen sei das lineare Gleichungssystem

$$Ax = b \tag{28}$$

mit einer $N \times N$-Matrix A und entsprechenden Vektoren x und b.
Gauß'sches Eliminationsverfahren (Sequentielles Verfahren)

```
(1) for (k = 0; k < N; k++)
(2)     for (i = k + 1; i < N; i++) {
(3)         l_ik = a_ik / a_kk;
(4)         for (j = k + 1; j < N; j++)
(5)             a_ij = a_ij - l_ik · a_kj;
(6)         b_i = b_i - l_ik · b_k;
        }
```



transformiert das Gleichungssystem (28) in das Gleichungssystem

$$Ux = d \tag{29}$$

mit einer oberen Dreiecksmatrix U.

Eigenschaften

Obige Formulierung hat folgende Eigenschaften:

- Die Matrixelemente a_{ij} für $j \geq i$ enthalten die entsprechenden Einträge von U, d.h. A wird überschrieben.
- Vektor b wird mit den Elementen von d überschrieben.
- Es wird angenommen, das a_{kk} in Zeile (3) immer von Null verschieden ist (keine Pivotisierung).

Ableitung aus Gauß-Elimination

Die LU-Zerlegung läßt sich aus der Gauß-Elimination ableiten:

- Jeder einzelne Transformationsschritt, der für festes k und i aus den Zeilen (3) bis (5) besteht, lässt sich als Multiplikation des Gleichungssystems mit einer Matrix schreiben:

$$\hat{L}_{ik} = \begin{array}{c} \\ \\ \\ \\ \text{i} \\ \\ \end{array} \begin{pmatrix} 1 & & \overset{k}{} & & & \\ & 1 & & & & \\ & & \ddots & & & \\ & & & \ddots & & \\ & & -l_{ik} & & \ddots & \\ & & & & & 1 \end{pmatrix} = I - l_{ik}E_{ik}$$

  (E_{ik} ist die Matrix deren Element $e_{ik} = 1$ ist, und die sonst aus lauter Nullen besteht) von links verstehen, mit l_{ik} aus Zeile (3) des Gauß'schen Eliminationsverfahrens.

- Somit gilt

$$\begin{aligned} &\hat{L}_{N-1,N-2} \cdot \ldots \cdot \hat{L}_{N-1,0} \cdot \ldots \cdot \hat{L}_{2,0} \cdot \hat{L}_{1,0} \cdot A \\ = &\hat{L}_{N-1,N-2} \cdot \ldots \cdot \hat{L}_{N-1,0} \cdot \ldots \cdot \hat{L}_{2,0} \cdot \hat{L}_{1,0} \cdot b \end{aligned} \tag{30}$$

und wegen (29) gilt

$$\hat{L}_{N-1,N-2} \cdot \cdots \cdot \hat{L}_{N-1,0} \cdot \cdots \cdot \hat{L}_{2,0} \cdot \hat{L}_{1,0} \cdot A = U. \tag{31}$$

Eigenschaften

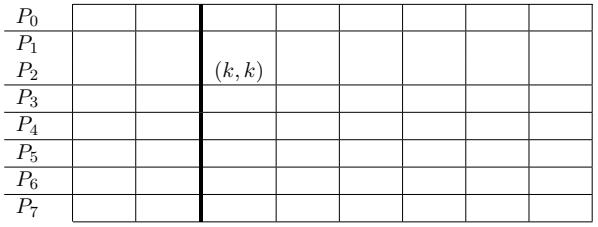
- Es gelten folgende Eigenschaften:
  1. $\hat{L}_{ik} \cdot \hat{L}_{i',k'} = I - l_{ik}E_{ik} - l_{i'k'}E_{i'k'}$ für $k \neq i'$ $(\Rightarrow E_{ik}E_{i'k'} = 0)$.
  2. $(I - l_{ik}E_{ik}) \cdot (I + l_{ik}E_{ik}) = I$ für $k \neq i$, d.h. $\hat{L}_{ik}^{-1} = I + l_{ik}E_{ik}$.
- Wegen 2 und der Beziehung (31)

$$A = \underbrace{\hat{L}_{1,0}^{-1} \cdot \hat{L}_{2,0}^{-1} \ldots \cdot \hat{L}_{N-1,0}^{-1} \cdot \ldots \cdot \hat{L}_{N-1,N-2}^{-1}}_{=:L} \cdot U = LU \tag{32}$$

- Wegen 1, was sinngemäß auch für $\hat{L}_{ik}^{-1} \cdot \hat{L}_{i'k'}^{-1}$ gilt, ist L eine untere Dreiecksmatrix mit $L_{ik} = l_{ik}$ für $i > k$ und $L_{ii} = 1$.
- Den Algorithmus zur LU-Zerlegung von A erhält man durch weglassen von Zeile (6) im Gauß-Algorithmus oben. Die Matrix L wird im unteren Dreieck von A gespeichert.

Parallele Variante mit zeilenweiser Aufteilung

Zeilenweise Aufteilung einer $N \times N$-Matrix für den **Fall** $N = P$:



- Im Schritt k teilt Prozessor P_k die Matrixelemente $a_{k,k}, \ldots, a_{k,N-1}$ allen Prozessoren P_j mit $j > k$ mit, und diese eliminieren in ihrer Zeile.
- Parallele Laufzeit:

$$\begin{aligned} T_P(N) &= \underbrace{\sum_{m=N-1}^{1}}_{\substack{\text{Anz. zu}\\ \text{eliminie-}\\ \text{render}\\ \text{Zeilen}}} (t_s + t_h + \underbrace{t_w \cdot m}_{\substack{\text{Rest der}\\ \text{Zeile } k}}) \underbrace{\operatorname{ld} N}_{\text{Broadcast}} + \underbrace{m \cdot 2t_f}_{\text{Elimination}} \\ &= \frac{(N-1) \cdot N}{2} 2t_f + \frac{(N-1) \cdot N}{2} \operatorname{ld} N \cdot t_w + N \operatorname{ld} N \cdot (t_s + t_h) \\ &\approx N^2 t_f + N^2 \operatorname{ld} N \frac{t_w}{2} + N \operatorname{ld} N \cdot (t_s + t_h) \end{aligned} \tag{33}$$

Analyse der parallelen Variante

- Sequentielle Laufzeit der LU-Zerlegung:

$$T_S(N) = \sum_{m=N-1}^{1} \underbrace{m}_{\substack{\text{Zeilen sind}\\ \text{zu elim.}}} \underbrace{m \cdot 2t_f}_{\substack{\text{Elim. einer}\\ \text{Zeile}}} = \tag{34}$$
$$= 2t_f \frac{(N-1)\cdot N \cdot (2\cdot(N-1)+1)}{6} \approx \frac{2}{3}N^3 t_f.$$

- Wie man aus (33) sieht, wächst $N \cdot T_P = \Omega(N^3 \operatorname{ld} N)$ (beachte $P = N!$) asymptotisch schneller als $T_S = \mathcal{O}(N^3)$.
- Der Algorithmus ist also nicht kostenoptimal (Effizienz kann für $P = N \longrightarrow \infty$ nicht konstant gehalten werden).
- Dies liegt daran, dass Prozessor P_k in seinem Broadcast wartet, bis alle anderen Prozessoren die Pivotzeile erhalten haben.
- Wir betrachten nun eine *asynchrone* Variante, bei der ein Prozessor sofort losrechnet, sobald er die Pivotzeile erhalten hat.

Asynchrone Variante

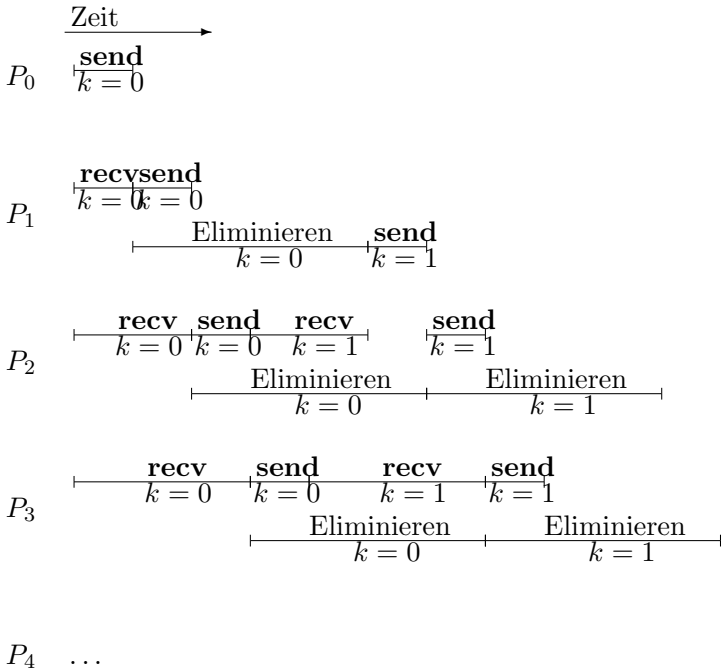
Programm 15.3 (Asynchrone LU-Zerlegung für $P = N$)**.**

```
parallel lu-1
{
    const int N = ...;
    process Π[int p ∈ {0,...,N−1}]
    {
        double A[N];                    // meine Zeile
        double rr[2][N];                // Puffer für Pivotzeile
        double *r;
        msgid m;
        int j, k;

        if (p > 0) m =arecv(Π_{p−1},rr[0]);
        for (k = 0; k < N − 1; k++)
        {
            if (p == k) send(Π_{p+1},A);
            if (p > k)
            {
                while (¬success(m));    // warte auf Pivotzeile
                if (p < N − 1) asend(Π_{p+1},rr[k%2]);
                if (p > k + 1) m =arecv(Π_{p−1},rr[(k + 1)%2]);
                r = rr[k%2];
                A[k] = A[k]/r[k];
                for (j = k + 1; j < N; j++)
```

```
                A[j] = A[j] - A[k] · r[j];
            }
        }
    }
}
```

Zeitlicher Ablauf

Wie verhält sich der parallele Algorithmus über die Zeit?



Parallele Laufzeit und Effizienz

- Nach einer Einlaufzeit von P Nachrichtenübertragungen ist die Pipeline gefüllt, und alle Prozessoren sind ständig mit eliminieren beschäftigt. Somit erhält man folgende Laufzeit ($N = P$, immer noch!):

$$
\begin{aligned}
T_P(N) &= \underbrace{(N-1)\cdot(t_s+t_h+t_wN)}_{\text{Einlaufzeit}} + \sum_{m=N-1}^{1} (\underbrace{m\cdot 2t_f}_{\text{Elim.}} + \underbrace{t_s}_{\substack{\text{Aufsetzzeit}\\ \text{(Rechnen+}\mathbf{send}\text{ parallel)}}}) = \qquad (35)\\
&= \frac{(N-1)\cdot N}{2} 2t_f + (N-1)\cdot(2t_s+t_h) + N\cdot(N-1)\cdot t_w \approx \\
&\approx N^2 t_f + N^2 t_w + N(2t_s+t_h).
\end{aligned}
$$

- Den Faktor $\mathrm{ld}\, N$ von (33) sind wir somit los. Für die Effizienz erhalten wir

$$\begin{aligned} E(N,P) &= \frac{T_S(N)}{NT_P(N,P)} = \frac{\frac{2}{3}N^3 t_f}{N^3 t_f + N^3 t_w + N^2(2t_s + t_h)} = \\ &= \frac{2}{3}\frac{1}{1 + \frac{t_w}{t_f} + \frac{2t_s + t_h}{N t_f}}. \end{aligned} \tag{36}$$

- Die Effizienz ist also maximal $\frac{2}{3}$. Dies liegt daran, dass Prozessor k nach k Schritten idle bleibt. Verhindern lässt sich dies durch mehr Zeilen pro Prozessor (gröbere Granularität).

Der Fall $N \gg P$

LU-Zerlegung für den Fall $N \gg P$:

- Programm 15.3 von oben lässt sich leicht auf den Fall $N \gg P$ erweitern. Dazu werden die *Zeilen* zyklisch auf die Prozessoren $0, \ldots, P-1$ verteilt. Die aktuelle Pivotzeile erhält ein Prozessor vom Vorgänger im Ring.

- Die parallele Laufzeit ist

$$\begin{aligned} T_P(N,P) &= \underbrace{(P-1)\cdot(t_s + t_h + t_w N)}_{\substack{\text{Einlaufzeit der} \\ \text{Pipeline}}} + \sum_{m=N-1}^{1} \Big(\underbrace{\frac{m}{P}}_{\substack{\text{Zeilen} \\ \text{pro} \\ \text{Prozes-} \\ \text{sor}}} \cdot m \cdot 2t_f + t_s \Big) = \\ &= \frac{2}{3}\frac{N^3}{P} \cdot t_f + N \cdot t_s + P \cdot (t_s + t_h) + NP \cdot t_w \end{aligned}$$

  und somit hat man die Effizienz

$$E(N,P) = \frac{1}{1 + \frac{P t_s}{N^2 \frac{2}{3} t_f} + \ldots}.$$

- Wegen der zeilenweisen Aufteilung gilt jedoch in der Regel, dass einige Prozessoren eine Zeile mehr haben als andere.

- Eine noch bessere Lastverteilung erzielt man durch eine zweidimensionale Verteilung der Matrix. Dazu nehmen wir an, dass die Aufteilung der Zeilen- und Spaltenindexmenge

$$I = J = \{0, \ldots, N-1\}$$

  durch die Abbildungen p und μ für I und q und ν für J vorgenommen wird.

Allgemeine Aufteilung

- Die nachfolgende Implementierung wird vereinfacht, wenn wir zusätzlich noch annehmen, dass die Datenverteilung folgende Monotoniebedingung erfüllt:

$$\begin{array}{lll} \text{Ist } i_1 < i_2 \text{ und } p(i_1) = p(i_2) & \text{so gelte} & \mu(i_1) < \mu(i_2) \\ \text{ist } j_1 < j_2 \text{ und } q(j_1) = q(j_2) & \text{so gelte} & \nu(j_1) < \nu(j_2) \end{array}$$

- Damit entspricht einem Intervall von globalen Indizes $[i_{min}, N-1] \subseteq I$ eine Anzahl von Intervallen lokaler Indizes in verschiedenen Prozessoren, die wie folgt berechnet werden können:

$$\begin{aligned} & \text{Setze} \\ \tilde{I}(p,k) &= \{m \in \mathbb{N} \mid \exists i \in I, i \geq k \colon p(i) = p \wedge \mu(i) = m\} \\ & \text{und} \\ \mathit{ibegin}(p,k) &= \begin{cases} \min \tilde{I}(p,k) & \text{falls } \tilde{I}(p,k) \neq \emptyset \\ N & \text{sonst} \end{cases} \\ \mathit{iend}(p,k) &= \begin{cases} \max \tilde{I}(p,k) & \text{falls } \tilde{I}(p,k) \neq \emptyset \\ 0 & \text{sonst.} \end{cases} \end{aligned}$$

- Dann kann man eine Schleife

  for $(i = k;\ i < N;\ i++)\ \ldots$

  ersetzen durch lokale Schleifen in den Prozessoren p der Gestalt

  for $(i = \mathit{ibegin}(p,k);\ i \leq \mathit{iend}(p,k);\ i++)\ \ldots$

Analog verfährt man mit den Spaltenindizes:

$$\begin{aligned} & \text{Setze} \\ \tilde{J}(q,k) &= \{n \in \mathbb{N} \mid \exists j \in j, j \geq k \colon q(j) = q \wedge \nu(j) = n\} \\ & \text{und} \\ \mathit{jbegin}(q,k) &= \begin{cases} \min \tilde{J}(q,k) & \text{falls } \tilde{J}(q,k) \neq \emptyset \\ N & \text{sonst} \end{cases} \\ \mathit{jend}(q,k) &= \begin{cases} \max \tilde{J}(q,k) & \text{falls } \tilde{J}(q,k) \neq \emptyset \\ 0 & \text{sonst.} \end{cases} \end{aligned}$$

Damit können wir zur Implementierung der LU-Zerlegung für eine allgemeine Datenaufteilung schreiten.

Programm 15.4 (Synchrone LU-Zerlegung mit allg. Datenaufteilung)**.**

```
parallel lu-2
{
    const int N = ..., √P = ...;

    process Π[int (p,q) ∈ {0,...,√P − 1} × {0,...,√P − 1}]
    {
        double A[N/√P][N/√P], r[N/√P], c[N/√P];
        int i, j, k;

        for (k = 0; k < N − 1; k++)
        {
            I = μ(k); J = ν(k);              // lokale Indizes
```

```
        // verteile Pivotzeile:
        if (p == p(k))
        {                                   // Ich habe Pivotzeile
            for (j = jbegin(q,k); j ≤ jend(q,k); j++)
                r[j] = A[I][j];             // kopiere Segment der Pivotzeile
            Sende r an alle Prozessoren (x,q) ∀x ≠ p
        }
        else recv(Π_{p(k),q}, r);

    }
```

Programm 15.5 (Synchrone *LU*-Zerlegung mit allg. Datenaufteilung (ctd.)).

```
parallel lu-2 (ctd.)
{
    // verteile Pivotspalte:
        if (q == q(k))
        {                                   // Ich habe Teil von Spalte k
            for (i = ibegin(p,k+1); i ≤ iend(p,k+1); i++)
                c[i] = A[i][J] = A[i][J]/r[J];
            Sende c an alle Prozessoren (p,y) ∀y ≠ q
        }
        else recv(Π_{p,q(k)}, c);

        // Elimination:
        for (i = ibegin(p,k+1); i ≤ iend(p,k+1); i++)
            for (j = jbegin(q,k+1); j ≤ jend(q,k+1); j++)
                A[i][j] = A[i][j] - c[i] · r[j];
    }
}
```

- Analysieren wir diese Implementierung (synchrone Variante):

$$T_P(N,P) = \sum_{m=N-1}^{1} \underbrace{\left(t_s + t_h + t_w \frac{m}{\sqrt{P}}\right) \operatorname{ld}\sqrt{P}\, 2}_{\substack{\text{Broadcast} \\ \text{Pivotzeile/-} \\ \text{spalte}}} + \left(\frac{m}{\sqrt{P}}\right)^2 2t_f =$$

$$= \frac{N^3}{P}\frac{2}{3}t_f + \frac{N^2}{\sqrt{P}} \operatorname{ld}\sqrt{P} t_w + N \operatorname{ld}\sqrt{P}\, 2(t_s + t_h).$$

- Mit $W = \frac{2}{3}N^3 t_f$, d.h. $N = \left(\frac{3W}{2t_f}\right)^{\frac{1}{3}}$, gilt

$$T_P(W,P) = \frac{W}{P} + \frac{W^{\frac{2}{3}}}{\sqrt{P}} \operatorname{ld}\sqrt{P} \frac{3^{2/3} t_w}{(2t_f)^{\frac{2}{3}}} + W^{\frac{1}{3}} \operatorname{ld}\sqrt{P} \frac{3^{1/3} 2(t_s + t_h)}{(2t_f)^{\frac{1}{3}}}.$$

- Die Isoeffizienzfunktion erhalten wir aus $PT_P(W,P) - W \stackrel{!}{=} KW$:

$$\begin{aligned} & \sqrt{P}W^{\frac{2}{3}} \operatorname{ld}\sqrt{P}\frac{3^{2/3}t_w}{(2t_f)^{\frac{2}{3}}} = KW \\ \iff \quad & W = P^{\frac{3}{2}}(\operatorname{ld}\sqrt{P})^3\frac{9t_w^3}{4t_f^2K^3} \end{aligned}$$

  also

$$W \in O(P^{3/2}(\operatorname{ld}\sqrt{P})^3).$$

- Programm 15.4 kann man auch in einer asynchronen Variante realisieren. Dadurch können die Kommunikationsanteile wieder effektiv hinter der Rechnung versteckt werden.

Pivotisierung

- Die LU-Faktorisierung allgemeiner, invertierbarer Matrizen erfordert Pivotisierung und ist auch aus Gründen der Minimierung von Rundungsfehlern sinnvoll.

- Man spricht von voller Pivotisierung, wenn das Pivotelement im Schritt k aus allen $(N-k)^2$ verbleibenden Matrixelementen ausgewählt werden kann, bzw. von teilweiser Pivotisierung (*engl.* „partial pivoting“), falls das Pivotelement nur aus einem Teil der Elemente ausgewählt werden darf. Üblich ist z.B. das maximale Zeilen- oder Spaltenpivot, d.h. man wählt a_{ik}, $i \geq k$, mit $|a_{ik}| \geq |a_{mk}| \quad \forall m \geq k$.

- Die Implementierung der LU-Zerlegung muss nun die Wahl des neuen Pivotelements bei der Elimination berücksichtigen. Dazu hat man zwei Möglichkeiten:
  - Explizites Vertauschen von Zeilen und/oder Spalten: Hier läuft der Rest des Algorithmus dann unverändert (bei Zeilenvertauschungen muss auch die rechte Seite permutiert werden).
  - Man bewegt die eigentlichen Daten nicht, sondern merkt sich nur die Vertauschung von Indizes (in einem Integer-Array, das alte Indizes in neue umrechnet.

- Die parallelen Versionen besitzen unterschiedliche Eignung für Pivotisierung. Folgende Punkte sind für die parallele LU-Zerlegung mit teilweiser Pivotisierung zu bedenken:
  - Ist der Bereich in dem das Pivotelement gesucht wird in einem einzigen Prozessor (z.B. zeilenweise Aufteilung mit maximalem Zeilenpivot) gespeichert, so ist die Suche sequentiell durchzuführen. Im anderen Fall kann auch sie parallelisiert werden.
  - Allerdings erfordert diese parallele Suche nach dem Pivotelement natürlich Kommunikation (und somit Synchronisation), die das Pipelining in der asynchronen Variante unmöglich macht.
  - Permutieren der Indizes ist schneller als explizites Vertauschen, insbesondere, wenn das Vertauschen den Datenaustausch zwischen Prozessoren erfordert. Allerdings kann dadurch eine gute Lastverteilung zerstört werden, falls zufällig die Pivotelemente immer im gleichen Prozessor liegen.

- Einen recht guten Kompromiss stellt die zeilenweise zyklische Aufteilung mit maximalem Zeilenpivot und explizitem Vertauschen dar, denn:
  - Pivotsuche in der Zeile k ist zwar sequentiell, braucht aber nur $O(N-k)$ Operationen (gegenüber $O((N-k)^2/P)$ für die Elimination); ausserdem wird das Pipelining nicht gestört.
  - explizites Vertauschen erfordert nur Kommunikation des Index der Pivotspalte, aber keinen Austausch von Matrixelementen zwischen Prozessoren. Der Pivotspaltenindex wird mit der Pivotzeile geschickt.
  - Lastverteilung wird von der Pivotisierung nicht beeinflusst.

Lösen der Dreieckssysteme

- Wir nehmen an, die Matrix A sei in $A = LU$ faktorisiert wie oben beschrieben, und wenden uns nun der Lösung eines Systems der Form

$$LUx = b \tag{37}$$

  zu. Dies geschieht in zwei Schritten:

$$Ly = b \tag{38}$$
$$Ux = y. \tag{39}$$

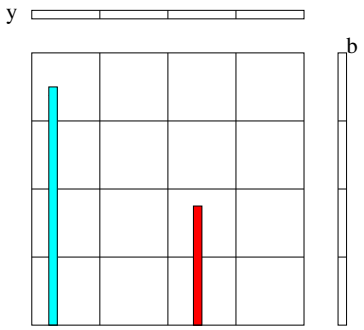
- Betrachten wir kurz den sequentiellen Algorithmus:

```
// Ly = b:
for (k = 0; k < N; k++) {
    y_k = b_k;
    for (i = k + 1; i < N; i++)
        b_i = b_i − a_ik y_k;
}
// Ux = y:
for (k = N − 1; k ≥ 0; k−−) {
    x_k = y_k / a_kk
    for (i = 0; i < k; i++)
        y_i = y_i − a_ik x_k;
}
```

- Dies ist eine spaltenorientierte Version, denn nach Berechnung von y_k bzw. x_k wird sofort die rechte Seite für alle Indizes $i > k$ bzw. $i < k$ modifiziert.

Parallelisierung

- Die Parallelisierung wird sich natürlich an der Datenverteilung der LU-Zerlegung orientieren müssen (falls man ein Umkopieren vermeiden will, was wegen $O(N^2)$ Daten und $O(N^2)$ Operationen sinnvoll erscheint). Betrachten wir hierzu eine zweidimensionale blockweise Aufteilung der Matrix:



- Die Abschnitte von b sind über Prozessorzeilen kopiert und die Abschnitte von y sind über Prozessorspalten kopiert. Offensichtlich können nach Berechnung von y_k nur die Prozessoren der Spalte $q(k)$ mit der Modifikation von b beschäftigt werden. Entsprechend können bei der Auflösung von $Ux = y$ nur die Prozessoren $(*, q(k))$ zu einer Zeit beschäftigt werden. Bei einer zeilenweise Aufteilung ($Q = 1$) sind somit immer alle Prozessoren beschäftigt.

Parallelisierung bei allg. Aufteilung

Programm 15.6 (Auflösen von $LUx = b$ bei allgemeiner Datenaufteilung)**.**

```
parallel lu-solve
{
    const int N = ...;
    const int √P = ...;
    process Π[int (p, q) ∈ {0, ..., √P − 1} × {0, ..., √P − 1}]
    {
        double A[N/√P][N/√P];
        double b[N/√P]; x[N/√P];
        int i, j, k, I, K;

        // Löse Ly = b, speichere y in x.
        // b spaltenweise verteilt auf Diagonalprozessoren.
        if (p == q) sende b an alle (p, *);
        for (k = 0; k < N; k++)
        {
            I = μ(k); K = ν(k);
            if(q(k) == q)                          // nur die haben was zu tun
            {
                if (k > 0 ∧ q(k) ≠ q(k − 1))       // brauche aktuelle b
                    recv(Π_{p,q(k−1)}, b);
                if (p(k) == p)
                {                                  // habe Diagonalelement
                    x[K] = b[I];                   // speichere y in x!
                    sende x[K] an alle (*, q);
                }
                else recv(Π_{p(k),q(k)}, x[k]);
                for (i = ibegin(p, k + 1); i ≤ iend(p, k + 1); i++)
```

```
                b[i] = b[i] − A[i][K] · x[K];
            if (k < N − 1 ∧ q(k + 1) ≠ q(k))
                send(Π_{p,q(k+1)}, b);
        }
    }
    ...

}
```

Programm 15.7 (Auflösen von $LUx = b$ bei allgemeiner Datenaufteilung cont.).

```
parallel lu-solve cont.
{

    ...
    // { y steht in x; x ist spaltenverteilt und zeilenkopiert. Für Ux = y wollen wir y in b
    //   speichern Es ist also x in b umzukopieren, wobei b zeilenverteilt und spaltenkopiert sein soll.
    for (i = 0; i < N/√P; i++)              // löschen
        b[i] = 0;
    for (j = 0; j < N − 1; j++)
        if (q(j) = q ∧ p(j) = p)            // einer muss es sein
            b[μ(j)] = x[ν(j)];
    summiere b über alle (p, ∗), Resultat in (p, p);

    // Auflösen von Ux = y (y ist in b gespeichert)
    if (p == q) sende b and alle (p, ∗);
    for (k = N − 1; k ≥ 0; k − −)
    {
        I = μ(k); K = ν(k);
        if (q(k) == q)
        {
            if (k < N − 1 ∧ q(k) ≠ q(k + 1))
                recv(Π_{p,q(k+1)}, b);
            if (p(k) == p)
            {
                x[K] = b[I]/A[I][K];
                sende x[K] an alle (∗, q);
            }
            else recv(Π_{p(k),q(k)}, x[K]);
            for (i = ibegin(p, 0); i ≤ iend(p, 0); i++)
                b[i] = b[i] − A[i][K] · x[K];
            if (k > 0 ∧ q(k) ≠ q(k − 1))
                send(Π_{p,q(k−1)}, b);
        }
    }
}
```

}

- Da zu einer Zeit immer nur $\sqrt{P}$ Prozessoren beschäftigt sind, kann der Algorithmus nicht kostenoptimal sein. Das Gesamtverfahren aus LU-Zerlegung und Auflösen der Dreieckssysteme kann aber immer noch isoeffizient skaliert werden, da die sequentielle Komplexität des Auflösens nur $O(N^2)$ gegenüber $O(N^3)$ für die Faktorisierung ist.
- Muss man ein Gleichungssystem für viele rechte Seiten lösen, sollte man ein rechteckiges Prozessorfeld $P \times Q$ mit $P > Q$ verwenden, oder im Extremfall $Q = 1$ wählen. Falls Pivotisierung erforderlich ist, war das ja ohnehin eine sinnvolle Konfiguration.

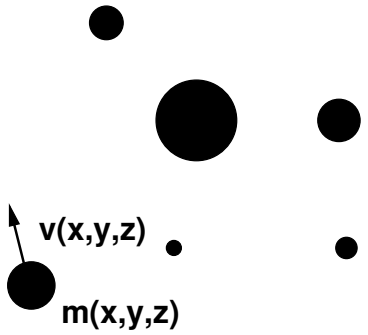
Standard Bibliotheken für Lineare Algebra

- Rein sequentiell
  - BLAS (Basic Linear Algebra Subprgrams)-Implementierungen, z.B. ATLAS, aber auch herstellerspezifische Varianten.
  - Lapack (Linear Algebra Package): Lineare Algebra für Fortran
  - Blitz++: Hoch-optimierte C++ Klassen mit Template-Metaprogramming und Expressiontemplates
- Parallel
  - ScaLaPack: Lapack für distributed memory Maschinen (Fortran)
  - Trilinos: Sammlung verschiedener Bibliotheken insbesondere für lineare Algebra (C,C++)
  - ISTL (Iterative Solver Template Library): Iterative Lösung dünnbesetzter linearer Gleichungssysteme (C++), Teil von DUNE
  - Hypre (High Performance Preconditioners): Bibliothek zur Lösung dünnbesetzter linearer Gleichungssysteme (C)
  - PETSc (Portable, Extensible Toolkit for Scientific Computation): Infrastruktur zur Lösung partieller Differentialgleichungen
  - SuperLU: Direkte Lösung linearer Gleichungssysteme, sequentielle und parallele Varianten
  - PARDISO: Direkte Lösung linearer Gleichungssysteme, sequentielle und parallele Varianten

16 Partikelmethoden

Aufgabenstellung:

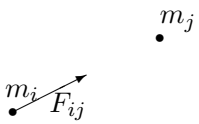
- Mit Partikelmethoden simuliert man die Bewegung von N Teilchen (oder Körpern) welche sich unter dem Einfluss eines Kraftfeldes bewegen.
- Das Kraftfeld selbst hängt wieder von der Position der Teilchen ab.
- Die Teilchen sind durch Punktmassen $m(x, y, z)$ charakterisiert und bewegen sich mit Geschwindigkeit $v(x, y, z)$ im System.



Das N-Körper-Problem

- Gegeben seien N punkförmige Massen $m_1, \ldots, m_N$ an den Positionen $x_1, \ldots, x_N \in \mathbb{R}^3$.
- Die von Körper j auf Körper i ausgeübte Gravitationskraft ist gegeben durch das Newton'sche Gravitationsgesetz

$$F_{ij} = \underbrace{\frac{\gamma\, m_i m_j}{\|x_j - x_i\|^2}}_{\text{Stärke}} \cdot \underbrace{\frac{x_j - x_i}{\|x_j - x_i\|}}_{\substack{\text{Einheitsvektor von} \\ x_i \text{ in Richtung } x_j}} \tag{40}$$



- Die Gravitationskraft lässt sich auch als Potentialgradient schreiben:

$$F_{ij} = m_i \frac{\gamma\, m_j (x_j - x_i)}{\|x_j - x_i\|^3} = m_i\, \nabla \left(\frac{\gamma\, m_j}{\|x_j - x_i\|} \right) = m_i\, \nabla \phi_j(x_i). \tag{41}$$

  $\phi_y(x) = \frac{\gamma\, m}{\|y - x\|}$ heißt Gravitationspotential der Masse m an der Position $y \in \mathbb{R}^3$.

- Die Bewegungsgleichungen der betrachteten N Körper erhält man aus dem Gesetz Kraft=Masse×Beschleunigu

  für $i = 1, \ldots, N$

$$m_i \frac{dv_i}{dt} = m_i\, \nabla \left(\sum_{j \neq i} \frac{\gamma\, m_j}{\|x_j - x_i\|} \right) \tag{42}$$

$$\frac{dx_i}{dt} = v_i \tag{43}$$

Dabei ist $v_i(t)\colon \mathbb{R} \to \mathbb{R}^3$ die Geschwindigkeit des Körpers i und $x_i(t)\colon \mathbb{R} \to \mathbb{R}^3$ die Position in Abhängigkeit von der Zeit.

- Wir erhalten also ein System gewöhnlicher Differentialgleichungen der Dimension $6N$ (drei Raumdimensionen) für dessen Lösung noch Anfangsbedingungen für Position und Geschwindigkeit erforderlich sind:

$$x_i(0) = x_i^0, \quad v_i(0) = v_i^0, \quad \text{für } i = 1, \ldots, N \tag{44}$$

Numerische Berechnung

- Die Integration der Bewegungsgleichungen (42) und (43) erfolgt numerisch, da nur für $N = 2$ geschlossene Lösungen möglich sind (Kegelschnitte, Kepler'sche Gesetze).
- Das einfachste Verfahren ist das explizite Euler-Verfahren:

$$\text{Diskretisierung in der Zeit: } \begin{aligned} t^k &= k \cdot \Delta t, \\ v_i^k &= v_i(t^k), \\ x_i^k &= x_i(t^k). \end{aligned}$$

$$\left.\begin{aligned} v_i^{k+1} &= v_i^k + \Delta t \cdot \nabla\left(\sum_{j \neq i} \frac{\gamma\, m_j}{\| \underbrace{x_j^k - x_i^k}_{\text{„explizit“}} \|}\right) \\ x_i^{k+1} &= x_i^k + \Delta t \cdot v_i^k \end{aligned}\right\} \text{ für } i = 1, \ldots, N \tag{45}$$

Problematik der Kraftauswertung

- Es gibt bessere Verfahren als das explizite Eulerverfahren, das nur von der Konvergenzordnung $\mathcal{O}(\Delta t)$ ist, z.B. das Verfahren von Heun. Insbesondere ist das explizite Eulerverfahren nur für kleine Zeitschritte stabil. Für die Diskussion der Parallelisierung spielt dies jedoch keine große Rolle, da die Struktur anderer Verfahren ähnlich ist.
- Das Problem der Kraftauswertung: In der Kraftberechnung hängt die Kraft auf einen Körper i von der Position *aller* anderen Körper $j \neq i$ ab. Der Aufwand für eine Kraftauswertung (die mindestens einmal pro Zeitschritt notwendig ist) steigt also wie $\mathcal{O}(N^2)$ an. In den Anwendungen kann $N = 10^6, \ldots, 10^9$ sein, was einen enormen Rechenaufwand bedeutet.
- Ein Schwerpunkt dieses Kapitels ist daher die Vorstellung verbesserter sequentieller Algorithmen zur schnellen Kraftauswertung. Diese berechnen die Kräfte näherungsweise mit dem Aufwand $\mathcal{O}(N \log N)$ (Es gibt auch Verfahren mit $\mathcal{O}(N)$ Komplexität, die wir aus Zeitgründen weglassen).

Parallelisierung des Standardverfahrens

- Der $\mathcal{O}(N^2)$-Algorithmus ist recht einfach zu parallelisieren. Jeder der P Prozessoren erhält $\frac{N}{P}$ Körper. Um die Kräfte für alle seine Körper zu berechnen, benötigt ein Prozessor alle anderen Körper. Dazu werden die Daten im Ring zyklisch einmal herumgeschoben.

- Analyse:

$$
\begin{aligned}
T_S(N) &= c_1 N^2 \\
T_P(N,P) &= c_1 P \Big(\underbrace{\frac{N}{P} \cdot \frac{N}{P}}_{\substack{\text{Interaktion} \\ \text{von} \\ \text{Block } p \\ \text{mit} \\ \text{Block } q}} \Big) + \underbrace{c_2 P \frac{N}{P}}_{\text{Kommunikation}} = \\
&= c_1 \frac{N^2}{P} + c_2 N \\
E(P,N) &= \frac{c_1 N^2}{\left(c_1 \frac{N^2}{P} + c_2 N \right) P} = \frac{1}{1 + \frac{c_2}{c_1} \cdot \frac{P}{N}}
\end{aligned}
$$

- konstante Effizienz für $N = \Theta(P)$.
- Damit ist die Isoeffizienzfunktion wegen $W = c_1 N^2$

  $$W(P) = \Theta(P^2)$$

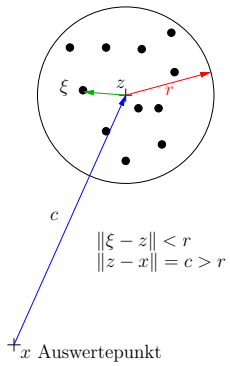
  (natürlich in Bezug auf den suboptimalen Standardalgorithmus mit $\mathcal{O}(N^2)$ Komplexität in der Anzahl Teilchen).

16.1 Schnelle Multipolmethoden

- Die erste grundlegende Idee wurde von *Pincus und Scherega* 1977 veröffentlicht. Wesentliche Idee war die Darstellung einer Gruppe von Partikeln durch ein sogenanntes Pseudopartikel. Dieses repräsentiert die Gruppeneigenschaften und somit das resultierende Potential. Die Beziehung mit einer anderen Partikelgruppe kann dann mit einer einzigen Multipol-Entwicklung berechnet werden.
- Ein zweites Konzept ist die hierarchische Vergröberung des Raumes in separierte Teilgebiete.
- Beide Methoden wurden durch Appel, 1985 und Barnes und Hut, 1986 innerhalb eines Algorithmus zusammengefaßt. Der Aufwand beträgt $\mathcal{O}(N \cdot \log N)$.
- Die schnelle Multipol Methode wird 1987 von Greengard und Rokhlin veröffentlicht. Zu den beiden genannten Ideen führen sie noch eine lokale Entwicklung von Potentialen ein. In bestimmten Fällen, etwa bei gleichmäßiger Partikelverteilung, reduziert sich der Aufwand auf $\mathcal{O}(N)$.

Schnelle Summationsmethoden

- Wir betrachten den abgebildeten Cluster aus Körpern: M Massenpunkte seien in einem Kreis um z mit dem Radius r enthalten. Wir werten das Potential ϕ aller Massenpunkte im Punkt x mit $\|z - x\| = c > r$ aus.



- Betrachten wir zunächst einen Massenpunkt an der Position ξ mit $\|\xi - z\| < r$. Das Potential der Masse in ξ sei (der multiplikative Faktor γm wird vernachlässigt).

$$\phi_\xi(x) = \frac{1}{\|\xi - x\|} = f(\xi - x).$$

- Das Potential hängt nur vom Abstand $\xi - x$ ab.

- Nun fügen wir den Punkt z ein und entwickeln in eine Taylorreihe um $(z - x)$ bis zur Ordnung p (nicht mit Prozessor verwechseln):

$$\begin{aligned} f(\xi - x) &= f((z - x) + (\xi - z)) = \\ &= \sum_{|\alpha| \le p} \frac{D^\alpha f(z - x)}{|\alpha|!} (\xi - z)^{|\alpha|} \\ &\quad + \underbrace{\sum_{|\alpha| = p} \frac{D^\alpha f(z - x + \theta(\xi - z))}{|\alpha|!} (\xi - z)^{|\alpha|}}_{\text{Restglied}} \end{aligned}$$

  für ein $\theta \in [0, 1]$. Wichtig ist die Separation der Variablen x und ξ.

- Die Größe des Fehlers (Restglied) hängt von p, r und c ab.

- Wie kann die Reihenentwicklung benutzt werden, um die Potentialauswertung zu beschleunigen?

- Dazu nehmen wir an, dass eine Auswertung des Potentials der M Massen an N Punkten zu berechnen ist, was normalerweise $\mathcal{O}(M \cdot N)$ Operationen erforderlich macht.

- Für die Auswertung des Potentials an der Stelle x berechnen wir

$$\begin{aligned}
\phi(x) \quad &= \quad \sum_{i=1}^{M} \gamma\, m_i \phi_{\xi_i}(x) = \sum_{i=1}^{M} \gamma\, m_i f((z-x)+(\xi_i - z)) \\
&\underset{\substack{\text{(Taylorreihe}\\ \text{bis}\\ \text{Ordnung}\\ p)}}{\approx} \quad \sum_{i=1}^{M} \gamma\, m_i \sum_{|\alpha|\le p} \frac{D^\alpha f(z-x)}{|\alpha|!}(\xi_i - z)^{|\alpha|} = \\
&\underset{\substack{\text{(Summe}\\ \text{vertau-}\\ \text{schen)}}}{=} \quad \sum_{|\alpha|\le p} \frac{D^\alpha f(z-x)}{|\alpha|!} \Bigg(\underbrace{\sum_{i=1}^{M} \gamma\, m_i (\xi_i - z)^{|\alpha|}}_{\substack{=: M_\alpha,\\ \text{unabhängig}\\ \text{von } x!}} \Bigg) = \\
&= \quad \sum_{|\alpha|\le p} \frac{D^\alpha f(z-x)}{|\alpha|!} M_\alpha
\end{aligned}$$

- Die Berechnung der Koeffizienten M_α erfordert einmalig $\mathcal{O}(Mp^3)$ Operationen.
- Sind die M_α bekannt, so kostet eine Auswertung von $\phi(x)$ $\mathcal{O}(p^5)$ Operationen.
- Bei Auswertung an N Punkten erhält man also die Gesamtkomplexität $\mathcal{O}(Mp^3 + Np^5)$.
- Es ist klar, dass das so berechnete Potential nicht exakt ist. Der Algorithmus macht nur Sinn, wenn der Fehler so kontrolliert werden kann, dass er vernachlässigbar ist (z.B. kleiner als der Diskretisierungsfehler).
- Ein Kriterium zur Fehlerkontrolle gibt die Fehlerabschätzung:

$$\frac{\phi_\xi(x) - \sum\limits_{|\alpha|\le p} \frac{D^\alpha f(z-x)}{|\alpha|!}(\xi - z)^{|\alpha|}}{\phi_\xi(x)} \le c(2h)^{p+1},$$

  mit $\frac{r}{c} \le h < \frac{1}{4}$. Für den Fall $c > 4r$ reduziert sich der Fehler wie $(1/2)^{p+1}$.
- Die Näherung wird also umso genauer,
  - je kleiner $\frac{r}{c}$
  - je größer der Entwicklungsgrad p.

Gradientenberechnung

- Im N-Körper-Problem will man nicht das Potential, sondern die Kraft, also den Gradient des Potentials, ausrechnen.
- Dies geht mittels

$$\frac{\partial \phi(x)}{\partial x_{(j)}} \underset{\substack{\text{Reihenentw.}\\ \uparrow\text{Raumdimension}}}{\approx} \frac{\partial}{\partial x_{(j)}} \sum_{|\alpha|\le p} \frac{D^\alpha f(z-x)}{|\alpha|!} M_\alpha = \sum_{|\alpha|\le p} \frac{D^\alpha \partial_{x_{(j)}} f(z-x)}{|\alpha|!} M_\alpha$$

- Man muss also nur $D^\alpha \partial_{x_{(j)}} f(z-x)$ statt $D^\alpha f(z-x)$ berechnen.

- Eine von Physikern häufig verwendete Approximation des Gravitationspotentials ist eine Taylorentwicklung von

$$\phi(x) = \sum_{i=1}^{M} \frac{\gamma\, m_i}{\|(s-x)+(\xi_i - s)\|},$$

  wobei s der *Massenschwerpunkt* der M Massen ist (und nicht ein fiktiver Kreismittelpunkt).

- Die sogenannte Monopolentwicklung lautet

$$\phi(x) \approx \frac{\sum_{i=1}^{M} \gamma\, m_i}{\|s-x\|}$$

  (d.h. ein Körper der Masse $\sum m_i$ in s).

- Die Genauigkeit wird dann nur über das Verhältnis r/c gesteuert.

Multipolmethoden

- Taylorreihe sind nicht die einzige Möglichkeit einer Reihenentwicklung. Daneben gibt es für die in Betracht kommenden Potentiale $\frac{1}{\log(\|\xi-x\|)}$ (2D) und $\frac{1}{\|\xi-x\|}$ (3D) andere Entwicklungen, die sogenannten Multipolentwicklungen, die sich besser eignen.

- Für diese Reihen gelten bessere Fehlerabschätzungen in dem Sinne, dass sie die Form

$$\textit{Fehler} \leq \frac{1}{1-\frac{r}{c}} \left(\frac{r}{c}\right)^{p+1}$$

  haben und somit schon für $c > r$ erfüllt sind.

- Außerdem ist die Komplexität in Bezug auf p besser (p^2 in 2D, p^4 in 3D).

16.2 Verschieben einer Entwicklung

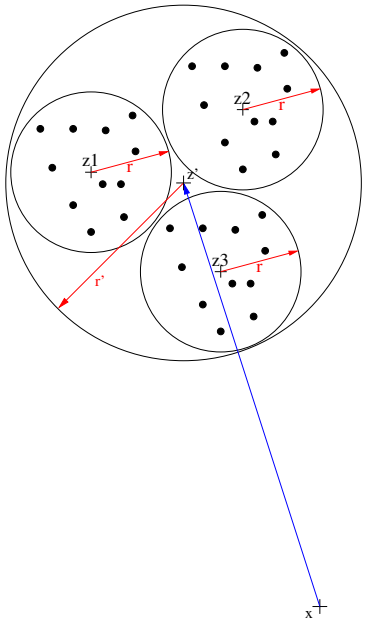
- In den unten folgenden Algorithmen benötigen wir noch ein Werkzeug, das die Verschiebung von Entwicklungen betrifft.

- Die Abbildung zeigt drei Cluster von Körpern in Kreisen um z_1, z_2, z_3 mit jeweils dem Radius r. Die drei Kreise sind in einem größeren Kreis um z' mit Radius r' enthalten.

- Wollen wir das Potential aller Massen in x mit $\|x - z'\| > r'$ auswerten, so könnten wir eine Reihenentwicklung um z' verwenden.

- Falls schon Reihenentwicklungen in den drei kleineren Kreisen berechnet wurden (d.h. die Koeffizienten M_α), so lassen sich die Entwicklungskoeffizienten der neuen Reihe aus denen der alten Reihen mit Aufwand $\mathcal{O}(p^\alpha)$ berechnen, d.h. unabhängig von der Zahl der Massen.

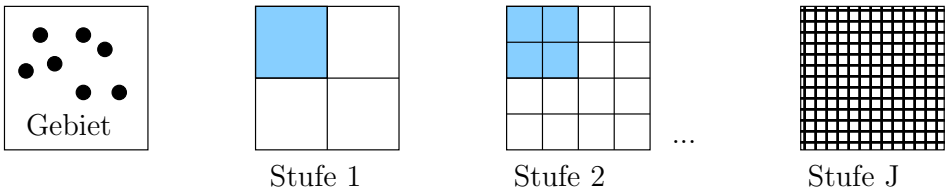
- Dabei entsteht *kein* zusätzlicher Fehler, und es gilt auch die Fehlerabschätzung mit entsprechend größerem r'.

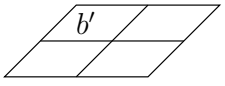


16.2.1 Gleichmäßige Punkteverteilung

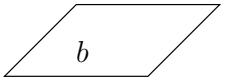
- Zunächst entwickeln wir einen Algorithmus, der sich für eine uniforme Verteilung der Punkte eignet. Dies hat den Vorteil einfacher Datenstrukturen und der Möglichkeit einfacher Lastverteilung. Wir stellen die Ideen für den zweidimensionalen Fall vor, da sich dies leichter zeichnen lässt. Alles kann jedoch in drei Raumdimensionen analog durchgeführt werden.
- Alle Körper seien in einem Quadrat $\Omega = (0, D_{max})^2$ der Seitenlänge D_{max} enthalten. Wir überziehen Ω mit einer Hierarchie von immer feineren Gittern. Stufe ℓ entsteht aus Stufe $\ell - 1$ durch Vierteln der Elemente.

Konstruktion der Gitter





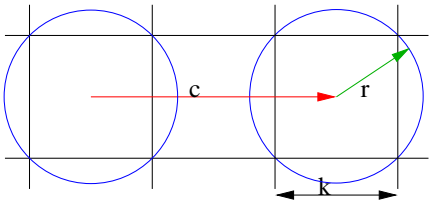
$\mathsf{Kinder}(b)$



$b = \mathsf{Vater}(b')$ für alle $b' \in \mathsf{Kinder}(b)$

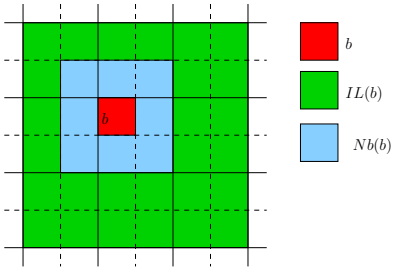
- Wollen wir s Körper pro Feingitterbox, so gilt $J = \log_4\left(\frac{N}{s}\right)$. Für zwei nicht benachbarte Quadrate erhalten wir folgende Abschätzung für das r/c-Verhältnis (Massen in b, Auswertung in a)

$$\begin{aligned} r &= \sqrt{2}\frac{k}{2} \\ c &= 2k \\ \Rightarrow \frac{r}{c} &= \frac{\sqrt{2}\,k}{4k} = \frac{\sqrt{2}}{4} \approx 0.35. \end{aligned}$$



r/c-Abschätzung für zwei nicht benachbarte Quadrate

- Für ein Element b auf Stufe ℓ definiert man folgende Bereiche in der Nachbarschaft von b:
  - $Nb(b)$ = alle *Nachbarn* b' von b auf Stufe ℓ ($\partial b \cap \partial b' \neq \emptyset$).
  - $IL(b)$ = *Interaktionsliste* von b: Kinder von Nachbarn von *Vater*(b), die nicht Nachbar von b sind.



Folgender Algorithmus berechnet das Potential an allen Positionen $x_1, \ldots, x_N$:

	Aufwand
1. Vorbereitungsphase:	
Für jede Feingitterbox berechne eine Fernfeldentwicklung;	$\mathcal{O}(\frac{N}{s}sp^\alpha)$
Für alle Stufen $\ell = J-1, \ldots, 0$	
Für jede Box b auf Stufe ℓ	
berechne Entwicklung in b aus Entwicklung in *Kinder*(b);	$\mathcal{O}(\frac{N}{s}p^\gamma)$
2. Auswertephase:	
Für jede Feingitterbox b	
Für jeden Körper q in b	
{	
berechne exaktes Potential aller $q' \in b$, $q' \neq q$;	$\mathcal{O}(\frac{N}{s}s^2)$
Für alle $b' \in Nb(b)$	
Für alle $q' \in b'$	
berechne Potential von q' in q;	$\mathcal{O}(8\frac{N}{s}s^2)$
$\bar{b} = b$;	
Für alle Stufen $\ell = J, \ldots, 0$	
{	
Für alle $b' \in IL(\bar{b})$	
Werte Fernfeldentwicklung von b' in q aus;	$\mathcal{O}(\frac{N}{s}sp^\beta)$
}	
}	

- Gesamtaufwand: $\mathcal{O}(\frac{N}{s}p^\gamma \log N + Ns + Np^\alpha + Np^\beta \log N)$, also asymptotisch $\mathcal{O}(N \log N)$.

- Dabei bezeichnet α den Exponenten für das Aufstellen der Fernfeldentwicklung und β den Exponenten für das Verschieben.

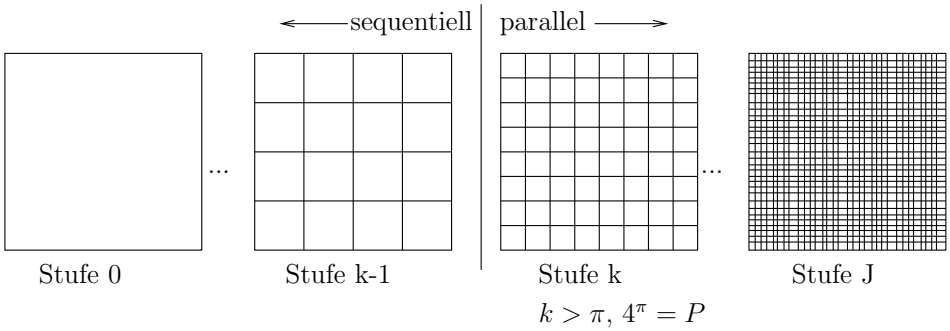
- Man beachte, dass man wegen der uniformen Verteilung N/s Körper pro Box auf Stufe J hat.

- Die Genauigkeit wird hier über den Entwicklungsgrad p gesteuert, das Verhältnis r/c ist fest.

Parallelisierung: Lastverteilung

- Lastverteilung: Das Gitter mit den zugehörigen Körpern wird auf die Prozessoren aufgeteilt.

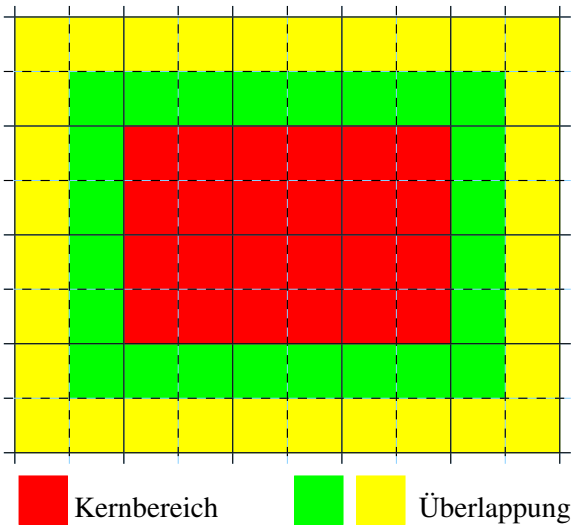
- Da wir nun eine Hierarchie von Gittern haben, verfahren wir wie folgt:
  - Jeder Prozessor soll mindestens 2×2 Gitterzellen bekommen.
  - Es sei $P = 4^\pi$, so wähle $k = \pi + 1$ und verteile das Gitter der Stufe k auf alle Prozessoren (ab hier hat jeder 2×2 Elemente).
  - Alle Stufen $\ell < k$ werden auf allen Prozessoren gespeichert.
  - Alle Stufen $\ell > k$ werden so aufgeteilt, dass $\mathit{Kinder}(b)$ im selben Prozessor wie b bearbeitet werden.
  - Beispiel für $P = 16 = 4^2$.



Aufteilung der Boxen bei der Parallelisierung

Parallelisierung: Überlappung

- Zusätzlich zu den ihm zugewiesenen Elementen speichert jeder Prozessor noch einen Überlappungsbereich von zwei Elementreihen:



Überlappungsbereich eines Prozessors

Parallelisierung: Analyse

- Da jeder mindestens 2×2 Elemente hat, liegt der Überlappungsbereich immer in direkten Nachbarprozessoren.

- Die Auswertung der *IL* erfordert höchstens Kommunikation mit nächsten Nachbarn.

- Zum Aufbau der Fernfeldentwicklung für die Stufen $\ell < k$ ist entweder eine alle-an-einen Kommunikation mit individuellen Nachrichten sowie eine einer-an-alle Kommunikation (wenn ein Rechner die Auswertung macht und dann wieder verteilt) oder eine alle-an-alle Kommunikation (wenn jeder für sich auswertet) erforderlich. Der Aufwand ist in beiden Fällen $\mathcal{O}(P + \log P)$

- Skalierbarkeitsabschätzung: Es sei $\frac{N}{sP} \gg 1$. Wegen

$$J = \log_4\left(\frac{N}{s}\right) = \log_4\left(\frac{N}{sP}P\right) = \underbrace{\log_4\left(\frac{N}{sP}\right)}_{J_p} + \underbrace{\log_4 P}_{J_s}$$

  werden J_s Stufen sequentiell und $J_p = \mathcal{O}(\frac{N}{P})$ Stufen parallel gerechnet.

- Somit erhalten wir für *festen Entwicklungsgrad:*

$$T_P(N,P) \underset{(\frac{N}{P}=const.)}{=} \underbrace{c_1\frac{N}{P}p^\alpha}_{\substack{\text{FFE in Fein-}\\\text{gitterboxen}}} + \underbrace{c_2\frac{Ns}{P}}_{\substack{\text{Nahfeld}\\\text{auswerten}}} + \underbrace{c_3 \operatorname{ld} P + c_4 P}_{\substack{\text{alle-an-alle. Es}\\\text{sind immer}\\\text{Daten für 4}\\\text{Zellen}}} + \underbrace{c_5\frac{N}{sP}J_p p^\gamma}_{\substack{\text{Verschieben}\\\text{FFE für}\\\ell = J\ldots k\\\text{parallel}}} + \underbrace{c_6 P J_s p^\gamma}_{\substack{\text{Verschieben}\\\text{FFE für}\\\ell = k-1\ldots 0\\\text{in } \textit{allen}\\\textit{Prozessoren}}} + \underbrace{c_7\frac{N}{P}J_p p^\beta}_{\substack{\text{Auswerten}\\\text{FFE}\\\text{Stufen}\\\ell \geq k}} + \underbrace{c_8\frac{N}{P}J_s p^\beta}_{\substack{\text{Auswerten}\\\text{FFE}\\\text{Stufen}\\\ell < k}}$$

- Also:

$$\begin{aligned} E(N,P) &= \frac{c_9 N \log N}{P\cdot\left[\left(\frac{c_5 p^\gamma}{s}+c_7 p^\beta\right)\frac{N}{P}\underbrace{J_p}_{\sim\log\frac{N}{P}\sim\log N} + \left(c_6 p^\gamma P + c_8\frac{N}{P}p^\beta\right)\underbrace{J_s}_{\log P} + \left(c_1 p^\alpha + c_2 s\right)\frac{N}{P} + c_4 P + c_3 \operatorname{ld} P\right]} \\ &\approx \frac{1}{b_1 + b_2\frac{\log P}{\log N} + b_2\frac{1}{\log N} + \frac{b_3 P^2 \log P + c_4 P^2 + c_3 P \operatorname{ld} P}{c_9 N \log N}} \end{aligned}$$

- Für eine isoeffiziente Skalierung muss N also etwa wie $P^2\frac{\log P}{\log N} \approx P^2$ wachsen!

16.2.2 Ungleichmäßige Verteilung

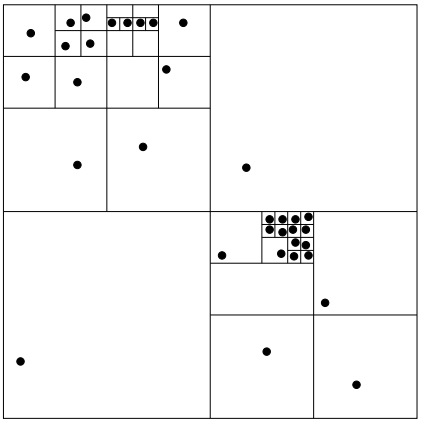
- Die Annahme einer uniformen Verteilung der Körper ist in einigen Anwendungen (z.B. Astrophysik) nicht realistisch.
- Will man in jeder Feingitterbox genau einen Körper haben und ist D_{min} der minimale Abstand zweier Körper, so braucht man ein Gitter mit

  $$\log L = \log \frac{D_{max}}{D_{min}}$$

  Gitterstufen. L heisst *„separation ratio“.*
- Von diesen sind aber die meisten leer. Wie bei dünnbesetzten Matrizen vermeidet man nun die leeren Zellen zu speichern. In zwei Raumdimensionen heisst diese Konstruktion „adaptiver Quadtree“ (in 3D entsprechend Octree).
- Der adaptive Quadtree wird folgendermaßen aufgebaut:
  - Initialisierung: Wurzel enthält alle Körper im Quadrat $(0, D_{max})$.
  - Solange es ein Blatt b mit mehr als zwei Körpern gibt:

  - Unterteile b in vier Teile
  - Packe jeden Körper von b in das entsprechende Kind
  - leere Kinder wieder wegwerfen.

- Beispiel eines adaptiven Quadtrees:

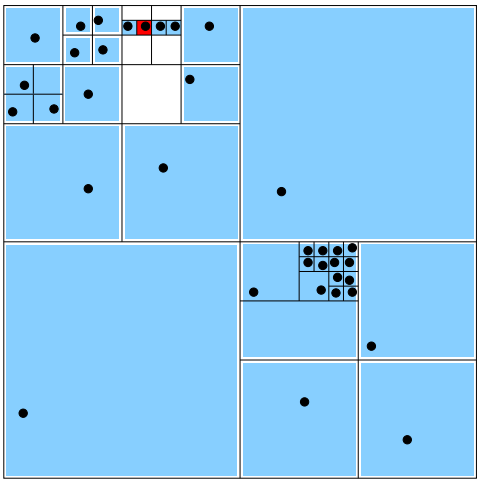


- Der Aufwand beträgt (sequentiell) $\mathcal{O}(N \log L)$.
- Der erste (erfolgreiche) schnelle Auswertealgorithmus für ungleichmäßig verteilte Körper wurde von Barnes und Hut vorgeschlagen.
- Wie im gleichmäßigen Fall wird eine Fernfeldentwicklung von den Blättern bis zur Wurzel aufgebaut (bei Barnes & Hut: Monopolentwicklung).
- Für einen Körper q berechnet dann folgender rekursive Algorithmus das Potential in q:

```
Pot(Körper q, Box b)
{
    double pot = 0;
    if (b ist Blatt ∧ q = b.q) return 0; // Ende
    if (Kinder(b) == ∅)
        return φ(b.q, q);                // direkte Auswertung
    if (r(b) / dist(q,b) ≤ h)
        return φ_b(q);                   // FFE Auswerten
    for (b' ∈ Kinder(b))
        pot = pot + Pot(q, b');          // rekursiver Abstieg
    return pot;
}
```

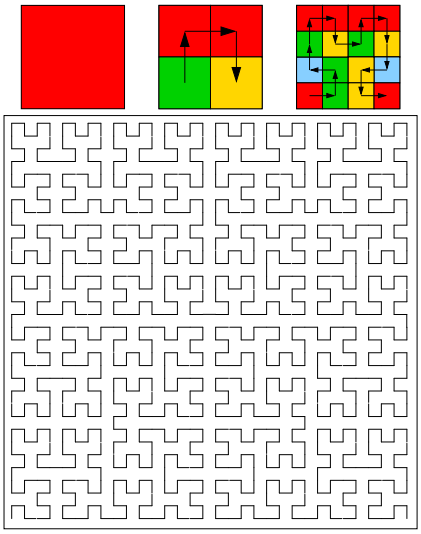
- Zur Berechnung wird Pot mit q und der Wurzel des Quadtree aufgerufen.
- Im Algorithmus von Barnes & Hut wird die Genauigkeit der Auswertung mittels des Parameters h gesteuert.

Welche Zellen des Quadtree werden im Barnes & Hut Algorithmus besucht?

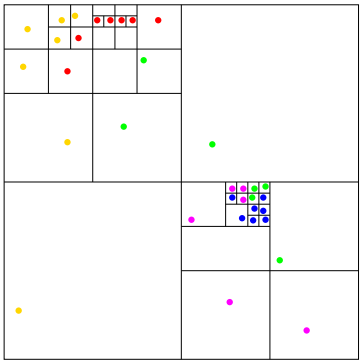

Beispiel zur Auswertung im Barnes & Hut Algorithmus

Parallelisierung

- Die Parallelisierung dieses Algorithmus ist relativ komplex, so dass wir nur einige Hinweise geben können. Für Details sei auf Salmon, 1994 verwiesen.
- Da sich die Positionen der Teilchen mit der Zeit verändern, muss der adaptive Quadtree in jedem Zeitschritt neu aufgebaut werden. Zudem muss man die Aufteilung der Körper auf die Prozessoren so vornehmen, dass nahe benachbarte Körper auch auf möglichst nahe benachbarten Prozessoren gespeichert sind.
- Ein sehr geschicktes Lastverteilungsverfahren arbeitet mit „raumfüllenden Kurven“. Die sogenannte Peano-Hilbert-Kurve hat folgende Gestalt:


Peano-Hilbert-Kurve

- Eine Hilbertkurve entsprechender Tiefe kann benutzt werden, um eine lineare Ordnung der Körper (bzw. der Blätter im Quadtree) herzustellen. Diese kann dann leicht in P Abschnitte der Länge N/P zerlegt werden.

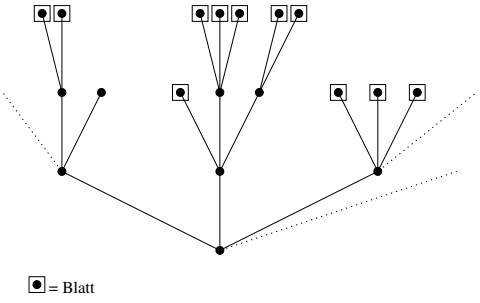


- Salmon & Warren zeigen, dass mit dieser Datenverteilung der adaptive Quadtree parallel mit sehr wenig Kommunikation aufgebaut werden kann. Ähnlich wie im uniformen Algorithmus wird dann durch eine alle-an-alle Kommunikation die Grobgitterinformation aufgebaut, die alle Prozessoren speichern.

Paralleler Aufbau des adaptiven Quadtree

- *Ausgangspunkt:* Jeder Prozessor hat eine Menge von Körpern, die *einem* zusammenhängenden Abschnitt auf der Hilbertkurve entspricht.

- *Schritt 1:* Jeder Prozessor baut lokal für *seine* Körper den Quadtree auf. Die „Nummern“ der Blätter sind dabei aufsteigend.



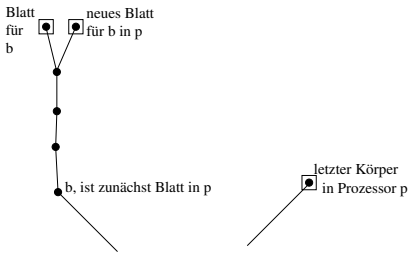
Lokaler Quadtree

- *Schritt 2:* Abgleich mit Nachbarprozessoren. Frage: Hätte ein sequentielles Programm für die Körper von Prozessor p dieselben Blätter erzeugt? Im allgemeinen nein, denn ein Körper von p und einer von $q \neq p$ könnten sich ja eine Box teilen.

- Was kann passieren?

  Sei b der *erste Körper* in Prozessor p und b' der *letzte* in Prozessor $p-1$ (entlang der Peano-Hilbert-Kurve). Nun gibt es zwei Möglichkeiten in Prozessor p:

  1. Körper b' liegt in der selben Box wie Körper b. $\Longrightarrow$ Zerteile Box so lange, bis beide Körper getrennt sind. Das neue Blatt ist das selbe, das auch ein sequentielles Programm berechnet hätte! Falls dem *nicht* so wäre, so müsste es einen Körper b'' in Prozessor $q \neq p$ geben, so dass $b'' \in$ neues Blatt von b. Dies ist aber unmöglich, da alle b'' vor b' oder nach dem letzten Knoten von Prozessor p kommen!



Austausch der Randblätter

2. Körper b' liegt nicht in derselben Box wie Körper b. $\Longrightarrow$ es ist nichts zu tun.

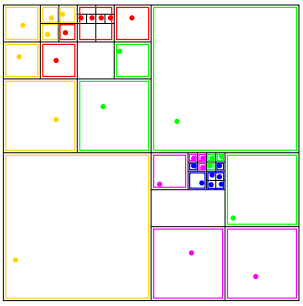
- Für den letzten Körper in p und den ersten in $p+1$ verfährt man ebenso!

Das Grobgitterproblem

- Wie im uniformen Fall wird der Quadtree von der Wurzel bis zu einer bestimmten Tiefe in jedem Prozessor gespeichert, so dass für diese Fernfeldauswertungen keine Kommunikation notwendig ist oder, falls nötig, der Prozessor bekannt ist, der über die entsprechende Information verfügt.

  Eine Box b im Quadtree heisst *Zweigknoten*, falls b nur Körper eines Prozessors p, der Vater von b jedoch Körper verschiedenener Prozessoren enthält. Diesem Prozessor p „gehört“ der Zweigknoten.

- Alle Boxen des Quadtree von der Wurzel bis einschließlich der Zweigknoten werden auf allen Prozessoren gespeichert.


Die Zweigknoten in unserem Beispiel

Kraftauswertung & Kommunikation

- Ist beim Auswerten ein rekursiver Aufruf in einem Zweigknoten nötig, so wird eine Nachricht an den entsprechenden Prozessor geschickt, dem der Zweigknoten gehört. Dieser bearbeitet die Anfrage asynchron und schickt das Ergebnis zurück.

- Nach dem Update der Positionen berechnet man eine neue Hilbertnummerierung (geht in $\frac{N}{P}\log L$ ohne Aufbau eines Quadtree) für die lokalen Körper. Dann bekommt jeder wieder einen Abschnitt der Länge $\frac{N}{P}$. Dies geht z.B. mit einem parallelen Sortieralgorithmus!

- Salmon & Warren konnten mit ihrer Implementierung bereits 1997 auf 6800 Prozessoren einer Intel ASCI-Red bis zu 322 Millionen Körper simulieren.

17 Parallele Sortierverfahren

- Es gibt eine ganze Reihe verschiedener Sortieralgorithmen: Heapsort, Bubblesort, Quicksort, Mergesort, ...
- Bei der Betrachtung paralleler Sortierverfahren beschränken wir uns auf
  - interne Sortieralgorithmen, d.h. solche, die ein Feld von (Ganz-) Zahlen im Speicher (wahlfreier Zugriff möglich!) sortieren, und
  - vergleichsbasierte Sortieralgorithmen, d.h. solche, bei denen die Grundoperationen aus Vergleich zweier Elemente und eventuellem Vertauschen besteht.
- Eingabe des parallelen Sortieralgorithmus besteht aus N Zahlen. Diese sind auf P Prozessoren verteilt, d.h. jeder Prozessor besitzt ein Feld der Länge N/P.
- Ergebnis eines parallelen Sortieralgorithmus ist eine sortierte Folge der Eingabezahlen, die wiederum auf die Prozessoren verteilt ist, d.h. Prozessor 0 enthält den ersten Block von N/P Zahlen, Prozessor 1 den zweiten usw.
- Wir behandeln zwei wichtige sequentielle Sortierverfahren: Mergesort und Quicksort.

17.1 Sequentielle Sortieralgorithmen

17.1.1 Mergesort

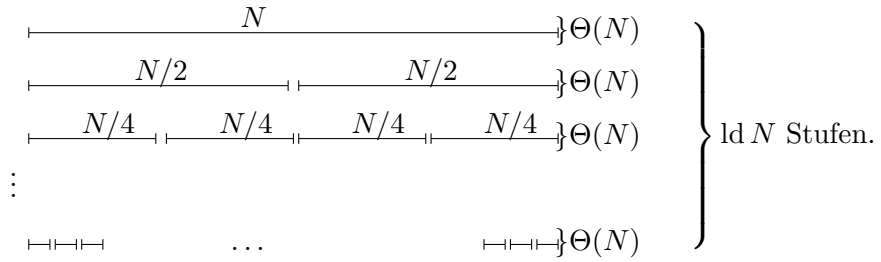
Mergesort basiert auf folgender Idee:

> Es sei eine Folge von N Zahlen zu sortieren. Angenommen wir teilen die Folge in zwei der Länge $\frac{N}{2}$ und sortieren diese jeweils getrennt, so können wir aus den beiden Hälften leicht eine sortierte Folge von N Zahlen erstellen, indem wir jeweils das nächste kleinste Element der Teilfolgen wegnehmen.

Algorithmus von Mergesort

Input: $a = \langle a_0, a_1, \ldots, a_{N-1} \rangle$;
$l = \langle a_0, \ldots, a_{N/2-1} \rangle$; $r = \langle a_{N/2}, \ldots, a_{N-1} \rangle$;
sortiere l; sortiere r;
$i = j = k = 0$;
while $(i < N/2 \ \wedge \ j < N/2)$
{
    if $(l_i \leq r_j)$
        $s_k = l_i$; $i++$;
    else
        $s_k = r_j$; $j++$;
    $k++$;
}
while $(i < N/2)$
{
    $s_k = l_i$; $i++$; $k++$;
}
while $(j < N/2)$
{
    $s_k = r_j$; $j++$; $k++$;
}

- Ein Problem von Mergesort ist, dass zusätzlicher Speicherplatz (zusätzlich zur Eingabe) erforderlich ist. Obige Variante kann sicher stark verbessert werden, „sortieren am Ort“, d.h. in a selbst, ist jedoch nicht möglich.
- *Laufzeit:* Das Mischen der zwei sortierten Folgen (die drei **while**-Schleifen) braucht $\Theta(N)$ Operationen. Die Rekursion endet in Tiefe $d = \operatorname{ld} N$, die Komplexität ist also $\Theta(N \operatorname{ld} N)$.



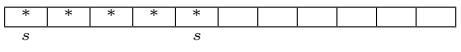
- Es zeigt sich, dass dies die optimale asymptotische Komplexität für vergleichsbasierte Sortieralgorithmen ist.

17.1.2 Quicksort

- Quicksort und Mergesort sind in gewissem Sinn komplementär zueinander.
- Mergesort sortiert rekursiv zwei (gleichgroße) Teilfolgen und mischt diese zusammen.
- Bei Quicksort zerteilt man die Eingabefolge in zwei Teilfolgen, so dass *alle* Elemente der ersten Folge kleiner sind als *alle* Elemente der zweiten Folge. Diese beiden Teilfolgen können dann getrennt (rekursiv) sortiert werden. Das Problem dabei ist, wie man dafür sorgt, dass die beiden Teilfolgen (möglichst) gleich groß sind, d.h. je etwa $N/2$ Elemente enthalten.
- Üblicherweise geht man so vor: Man wählt ein Element der Folge aus, z.B. das erste oder ein zufälliges und nennt es „Pivotelement“ Alle Zahlen kleiner gleich dem Pivotelement kommen in die erste Folge, alle anderen in die zweite Folge. Die Komplexität hängt nun von der Wahl des Pivotelementes ab:

$\Theta(N \log N)$	falls immer in zwei gleichgroße Mengen zerleg wird,
$\Theta(N^2)$	falls immer in eine einelementige Menge und de Rest zerlegt wird.

- Bei einer zufälligen Auswahl des Pivots ist der zweite Fall extrem unwahrscheinlich. In der Regel ist Quicksort schneller als Mergesort.



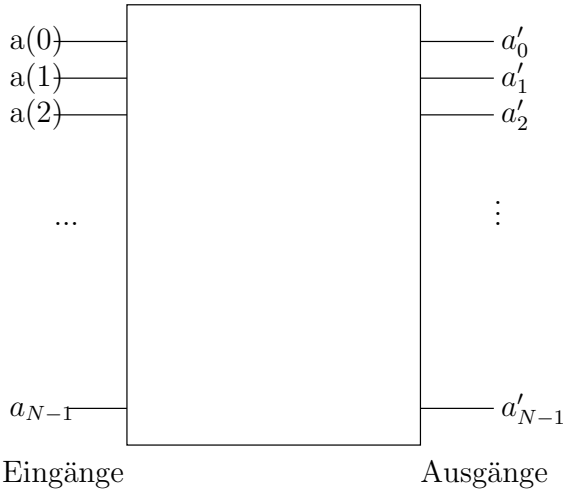

```
void Quicksort(int a[]; int first; int last)
{
    if (first ≥ last) return;                    // Ende der Rekursion
    // partitioniere
    wähle ein q ∈ [first, last];                 // Pivotwahl
    x = a[q];
    swap(a, q, first);                           // bringe x an den Anfang
    s = first;                                   // Marke Folge 1: [first ... s]
```

```
    for (i = first + 1; i ≤ last; i++)
        if (a[i] ≤ x)
        {
            s++;
            swap(a, s, i);                          // a[i] in erste Folge
        }
    swap(a, first, s);

    //          Nun gilt { 1. alle a[i] mit first ≤ i ≤ s sind ≤ x
    //                   { 2. a[s] = x ist bereits in der richtigen Position!

    Quicksort(a, first, s − 1);                     // a[s] wird nicht mitsortiert
    Quicksort(a, s + 1, last);                      // beide Folgen zusammen eins weniger
}
```

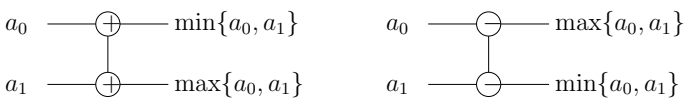
Stack könnte zum Problem werden, falls N groß und worst case erreicht wird (N rekursive Aufrufe).

17.2 Sortiernetzwerke

- Ein Sortiernetzwerk übersetzt eine Folge von N unsortierten Zahlen in eine Folge von N aufsteigend oder absteigend sortierten Zahlen.
- An den N Eingängen werden die unsortierten Zahlen angelegt, am Ausgang erscheint die sortierte Folge.

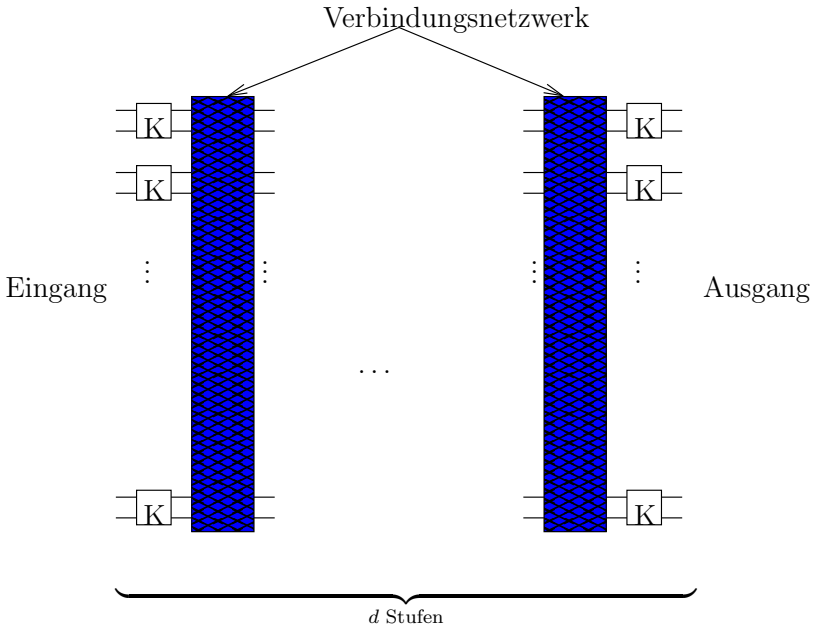


- Intern ist das Sortiernetzwerk aus elementaren Schaltzellen, sogenannten Komparatoren, aufgebaut, die genau zwei Zahlen aufsteigend oder absteigend sortieren.



- Eine Anzahl von Komparatoren wird zu einer sogenannten „Stufe“ zusammengefasst.

- Das ganze Netzwerk besteht aus mehreren Stufen, die ihrerseits durch ein Verbindungsnetzwerk verbunden sind.
- Die Anzahl der Stufen wird auch als Tiefe des Sortiernetzwerkes bezeichnet.



- Alle Komparatoren einer Stufe arbeiten parallel. Sortiernetzwerke können gut in Hardware realisiert werden oder lassen sich in entsprechende Algorithmen für Parallelrechner übertragen (weshalb wir sie hier studieren wollen).

17.3 Bitonisches Sortieren

Wir betrachten zunächst einen Algorithmus, der sowohl Eigenschaften von Quicksort als auch von Mergesort hat:

- Wie bei Mergesort wird die Eingabefolge der Länge N in zwei Folgen der Länge $N/2$ zerlegt. Dadurch ist die Rekursionstiefe immer ld N (allerdings wird die Gesamtkomplexität schlechter sein!)
- Wie bei Quicksort sind alle Elemente der einen Folge kleiner als alle Elemente der anderen Folge und können somit unabhängig voneinander sortiert werden.
- Das Zerlegen kann voll parallel mit $N/2$ Komparatoren geschehen, d.h. der Algorithmus kann mit einem Sortiernetzwerk realisiert werden.

17.3.1 Bitonische Folgen

Definition 17.1 (Bitonische Folge)**.** Eine Folge von N Zahlen heißt *bitonisch*, genau dann, wenn eine der beiden folgenden Bedingungen gilt

1. Es gibt einen Index $0 \leq i < N$, so dass

$$\underbrace{a_0 \leq a_1 \leq \cdots \leq a_i}_{\text{aufsteigend}} \text{ und } \underbrace{a_{i+1} \geq a_{i+2} \geq \cdots \geq a_{N-1}}_{\text{absteigend}}$$

2. Man kann die Indizes zyklisch schieben, d.h. $a'_i = a_{(i+m)\,\%\,N}$, so dass die Folge a' die Bedingung 1 erfüllt.

Normalform bitonischer Folgen

- Jede bitonische Folge lässt sich auf folgende Form bringen:

$$a'_0 \leq a'_1 \leq \cdots \leq \mathbf{a'_k} > \mathbf{a'_{k+1}} \geq a'_{k+2} \geq \cdots \geq \mathbf{a'_{N-1}} < \mathbf{a'_0}$$

- Wesentlich ist, dass das letzte Element des aufsteigenden Teils größer ist als der Anfang des absteigenden Teils.
- Ebenso ist das Ende des absteigenden Teils kleiner als der Anfang des aufsteigenden Teils.

Beweis
Für die Eingabefolge a gelte 1 aus der Definition der bitonischen Folgen. Es sei $a_0 \leq a_1 \leq \cdots \leq a_i$ der aufsteigende Teil. Entweder es gilt nun $a_i > a_{i+1}$ oder es gibt ein $j \geq 0$ mit

$$a_i \leq a_{i+1} = a_{i+2} = \cdots = a_{i+j} > a_{i+j+1}$$

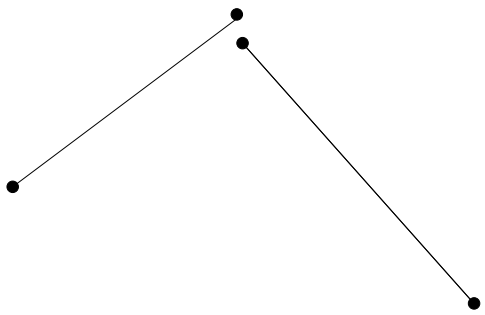
(oder die Folge ist trivial und besteht aus lauter gleichen Elementen). Somit kann man die $a_{i+1}, \ldots, a_{i+j}$ zum aufsteigenden Teil hinzunehmen und es gilt $k = i + j$.

Ebenso verfährt man mit dem absteigenden Teil. Entweder ist schon $a_{N-1} < a_0$ oder es gibt ein l, so dass

$$a_{N-1} \geq a_0 = a_1 = \cdots = a_l < a_{l+1}$$

(oder die Folge ist trivial). In diesem Fall nimmt man $a_0, \ldots, a_l$ zum absteigenden Teil hinzu.

- Man hat also die folgende Situation:



Min-Max Charakterisierung bitonischer Folgen

- In der folgenden Definition brauchen wir den Begriff „zyklischer Nachbar“, d.h. wir setzen

$$\begin{aligned} a_{i\oplus l} &:= a_{(i+l)\,\%\,N} \\ a_{i\ominus l} &:= a_{(i+N-l)\,\%\,N} \end{aligned}$$

für $l \leq N$

- Damit kommen wir zur

  Min-Max Charakterisierung bitonischer Folgen

  Definition 17.2. Ein a_k heisst (lokales) Maximum, falls es ein $l \geq 0$ gibt mit

  $$a_{k\ominus(l+1)} < a_{k\ominus l} = \cdots = a_{k\ominus 1} = a_k > a_{k\oplus 1}$$

  Entsprechend heisst a_k (lokales) Minimum, falls es ein $l \geq 0$ gibt mit

  $$a_{k\ominus(l+1)} > a_{k\ominus l} = \cdots = a_{k\ominus 1} = a_k < a_{k\oplus 1}$$

- Für nicht triviale Folgen (d.h. es sind nicht alle Elemente identisch) gilt damit: Eine Folge $a = \langle a_0, \ldots, a_{N-1}\rangle$ ist bitonisch genau dann, wenn sie genau ein Minimum und genau ein Maximum hat (oder trivial ist).

Beweis. $\Rightarrow$ gilt wegen der Normalform. a'_k ist das Maximum, a'_{N-1} das Minimum.

$\Leftarrow$ Wenn a genau ein Minimum bzw. Maximum hat, zerfällt sie in einen aufsteigenden und einen absteigenden Teil, ist also bitonisch.

□

17.3.2 Bitonische Zerlegung

Nach obigen Vorbereitungen können wir die Bitonischen Zerlegung charakterisieren

- Es sei $s = \langle a_0, \ldots, a_{N-1}\rangle$ eine gegebene bitonische Folge der Länge N.
- Wir konstruieren zwei neue Folgen der Länge $N/2$ (N gerade) auf folgende Weise:

  $$\begin{aligned} s_1 &= \left\langle \min\{a_0, a_{\frac{N}{2}}\}, \min\{a_1, a_{\frac{N}{2}+1}\}, \ldots, \min\{a_{\frac{N}{2}-1}, a_{N-1}\}\right\rangle \\ s_2 &= \left\langle \max\{a_0, a_{\frac{N}{2}}\}, \max\{a_1, a_{\frac{N}{2}+1}\}, \ldots, \max\{a_{\frac{N}{2}-1}, a_{N-1}\}\right\rangle \end{aligned} \tag{46}$$

  Für s_1, s_2 gilt:

  1. Alle Elemente von s_1 sind kleiner oder gleich allen Elementen in s_2.
  2. s_1 und s_2 sind bitonische Folgen.

- Offensichtlich können s_1 und s_2 aus s mit Hilfe von $N/2$ Komparatoren konstruiert werden.

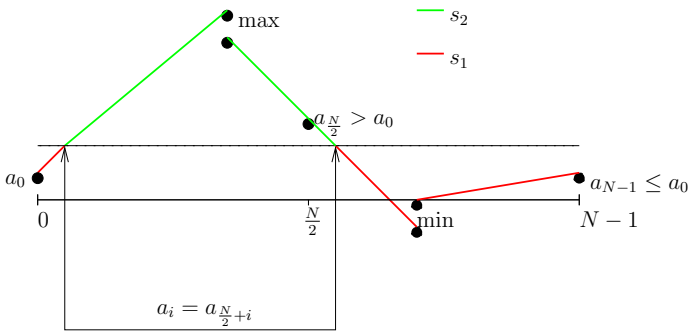
Beweis

Wir überlegen uns das graphisch anhand von verschiedenen Fällen:

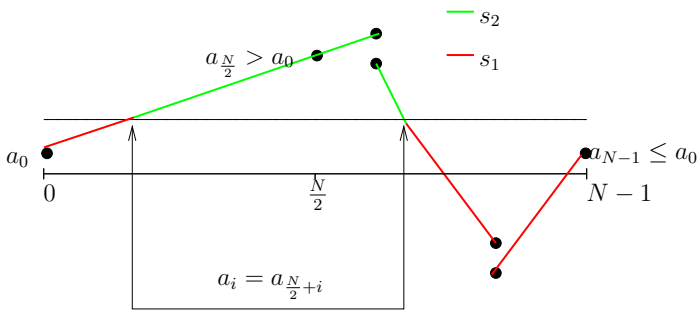
1. Es sei $a_0 < a_{\frac{N}{2}}$: Wir wissen, dass jede bitonische Folge genau ein Maximum und Minimum hat. Da $a_0 < a_{\frac{N}{2}}$ kann das Maximum nur zwischen a_0 und $a_{\frac{N}{2}}$ oder zwischen $a_{\frac{N}{2}}$ und a_{N-1} liegen.

   In diesem Fall gilt also $\min\{a_0, a_{\frac{N}{2}}\} = a_0$ und solange $a_i \leq a_{\frac{N}{2}+i}$ enthält s_1 die Elemente aus dem aufsteigenden Teil der Folge. Irgendwann gilt $a_i > a_{\frac{N}{2}+i}$, und dann enthält s_1 Elemente aus dem absteigenden Teil und anschließend wieder aus dem aufsteigenden Teil. Aus den folgenden Abbildungen ist sofort ersichtlich, dass s_1 und s_2 die Bedingungen 1 und 2 von oben erfüllen.

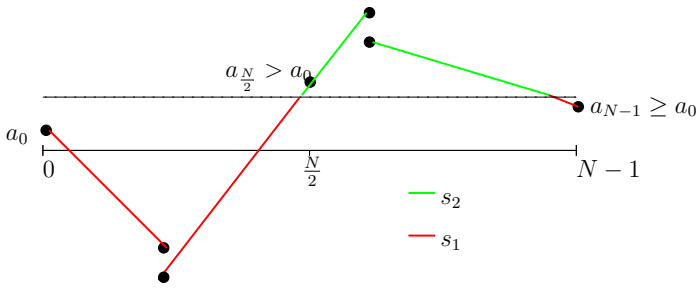
2. Maximum zwischen a_0 und $a_{\frac{N}{2}}$:



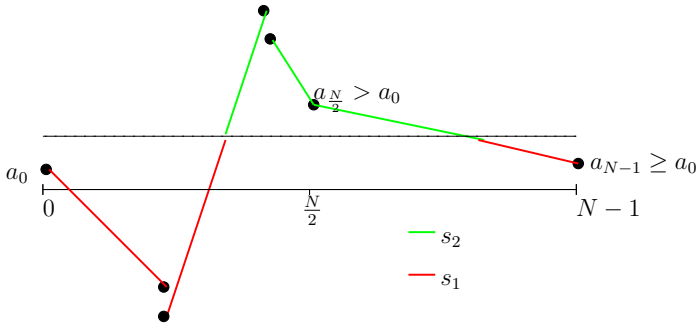
3. Maximum und Minimum zwischen $a_{\frac{N}{2}}$ und a_{N-1}:



4. Minimum zwischen a_0 und $a_{\frac{N}{2}}$:



5. Minimum und Maximum zwischen a_0 und $a_{\frac{N}{2}}$:

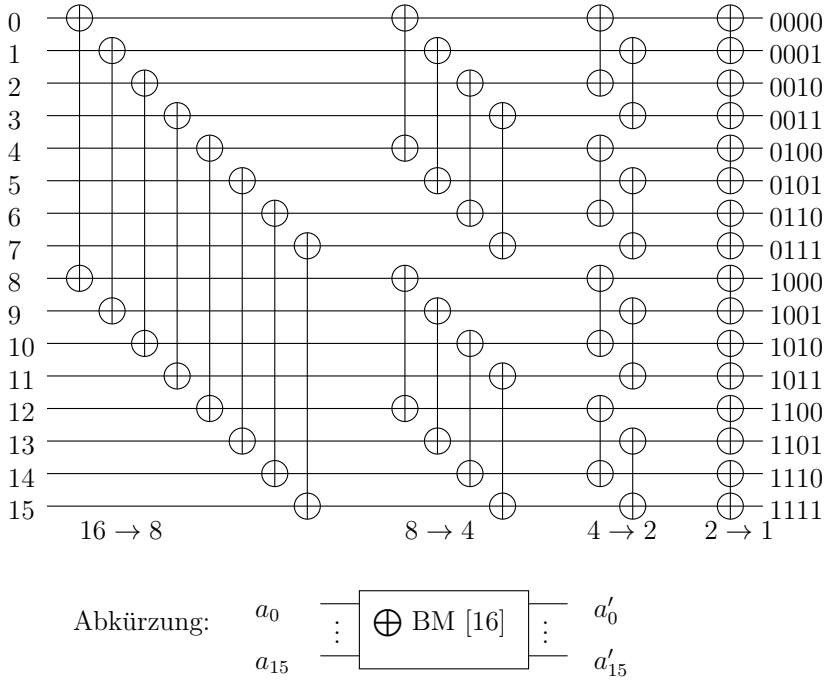


6. Die anderen Fälle: $a_0 = a_{\frac{N}{2}}$ bzw. $a_0 > a_{\frac{N}{2}}$ überlegt man sich analog.

17.3.3 Bitonischer Sortieralgorithmus

- Um eine *bitonische* Folge der Länge $N > 2$ zu sortieren, wendet man die bitonische Zerlegung rekursiv auf beide Hälften an. Die Rekursionstiefe ist natürlich d.

- Ein bitonisches Sortiernetzwerk, um eine bitonische Folge von 16 Zahlen zu sortieren, hat folgende Gestalt:

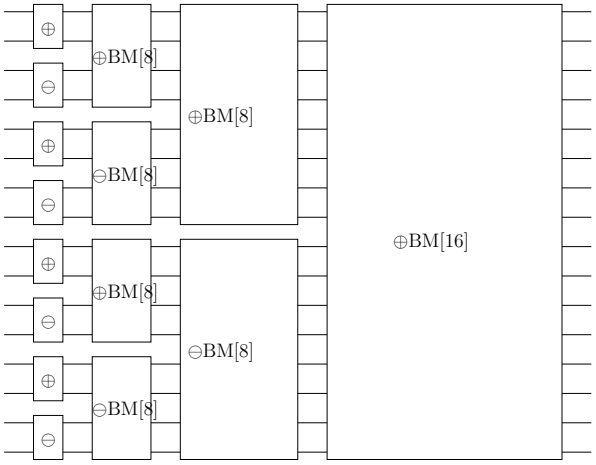


- Um nun N *unsortierte* Zahlen zu sortieren, muss man diese in eine bitonische Folge verwandeln.

- Wir bemerken zunächst, dass man durch bitonisches Zerlegen eine bitonische Folge auch leicht in absteigender Reihenfolge sortieren kann. Dazu ist in der bitonischen Zerlegung (46) max mit min zu vertauschen. Entsprechend sind im Netzwerk die $\oplus$-Komparatoren durch $\ominus$-Komparatoren zu ersetzen. Das entsprechende Netzwerk bezeichnet man mit $\ominus$BM[N].

Bitonische Folgen erzeugt man folgendermaßen:

- Eine Folge der Länge zwei ist immer bitonisch, da $a_0 \leq a_1$ oder $a_0 > a_1$. Hat man zwei bitonische Folgen der Länge N, so sortiert man die eine mit $\oplus$BM[N] aufsteigend und die andere mit $\ominus$BM[N] absteigend und erhält so eine bitonische Folge der Länge $2N$. Das vollständige Netzwerk zum Sortieren von 16 Zahlen sieht so aus:



Komplexität

- Betrachten wir die Komplexität des bitonischen Sortierens.
- Für die Gesamttiefe $d(N)$ des Netzwerkes bei $N = 2^k$ erhalten wir

$$\begin{aligned} d(N) &= \underbrace{\operatorname{ld} N}_{\oplus\mathrm{BM}[N]} + \underbrace{\operatorname{ld} \tfrac{N}{2}}_{\mathrm{BM}[N/2]} + \operatorname{ld} \tfrac{N}{4} + \ldots + \operatorname{ld} 2 = \\ &= k + k-1 + k-2 + \ldots + 1 = \\ &= \Theta(k^2) = \Theta(\operatorname{ld}^2 N). \end{aligned}$$

- Damit ist die Gesamtkomplexität für bitonisches Mischen also $\Theta(N \operatorname{ld}^2 N)$.

17.3.4 Bitonisches Sortieren auf dem Hypercube

- Wir überlegen nun, wie bitonisches Sortieren auf dem Hypercube realisiert werden kann.
- Dazu unterscheiden wir die Fälle $N = P$ (jeder hat eine Zahl) und $N \gg P$ (jeder hat einen Block von Zahlen).
- Der Fall $N = P$:

  Bitonisches Sortieren läst sich optimal auf den Hypercube abbilden! Am 4-stufigen Sortiernetzwerk erkennt man, daß nur nächste Nachbar Kommunikation erforderlich ist! Im Schritt $i = 0, 1, \ldots, d-1$ kommuniziert Prozessor p mit Prozessor $q = p \oplus 2^{d-i-1}$ (das $\oplus$ bedeutet hier wieder `XOR`). Offensichtlich erfordert aber jeder Vergleich eine Kommunikation, die Laufzeit wird also von den Aufsetzzeiten t_s der Nachrichten dominiert.
- Der Fall $N \gg P$:

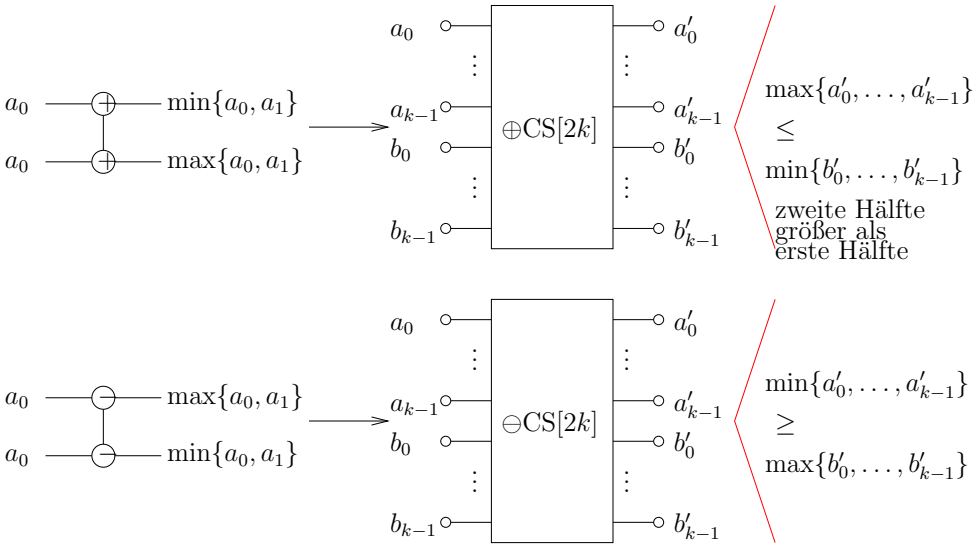
  Nun erhält jeder Prozessor einen Block von $K = \frac{N}{P}$ (N durch P teilbar) Zahlen.

  Nun kann man die Aufsetzzeiten amortisieren, da jeder Prozessor K Vergleiche durchführt, er „simuliert“ sozusagen K Komparatoren des Netzwerkes.
- Es bleibt noch das Problem, dass bitonisches Sortieren keine optimale sequentielle Komplexität besitzt, und somit keine isoeffiziente Skalierung erreicht werden kann.

- Dies kann man verbessern durch folgenden Ansatz:

  Es sei $P = 2^k$. Wir stellen uns bitonisches Sortieren auf P Elementen vor, nur dass jedes Element nun seinerseits eine Folge von K Zahlen ist.

- Es zeigt sich, dass man $P \cdot K$ Zahlen sortieren kann, indem man die Komparatoren für zwei Zahlen durch Bausteine für $2K$ Zahlen ersetzt, wie sie in folgendem Bild dargestellt sind:



- Dabei sind die Folgen $\langle a_0, \ldots, a_{k-1} \rangle$ bzw. $\langle b_0, \ldots, b_{k-1} \rangle$ bereits aufsteigend sortiert, und die Ausgabefolgen $\langle a'_0, \ldots, a'_{k-1} \rangle$ bzw. $\langle b'_0, \ldots, b'_{k-1} \rangle$ *bleiben* aufsteigend sortiert.

- CS[$2K$] kann auf zwei Prozessoren in $O(K)$ Zeit durchgeführt werden, indem jeder dem anderen seinen Block schickt, jeder die beiden Blöcke mischt und die richtige Hälfte behält.

- Wir sind also wieder bei einer Art Mergesort gelandet.

- Für die Gesamtkomplexität erhalten wir:

$$T_P(N,P) = c_1 \underbrace{\log^2 P}_{\text{Stufen}} \underbrace{\frac{N}{P}}_{\substack{\text{Aufwand}\\ \text{pro}\\ \text{Stufe}}} + \underbrace{c_2 \frac{N}{P} \log \frac{N}{P}}_{\substack{\text{einmaliges}\\ \text{Sortieren der}\\ \text{Eingabeblöcke}}} + \underbrace{c_3 \log^2 P \frac{N}{P}}_{\text{Kommunikation}}$$

- Somit gilt für die Effizienz

$$\begin{aligned} E(N,P) &= \frac{T_S}{T_P P} = \frac{c_2 N \log N}{\left(c_2 \frac{N}{P} \log \frac{N}{P} + (c_1 + c_3) \frac{N}{P} \log^2 P\right) P} = \\ &\underset{\substack{\log \frac{N}{P} = \\ \log N - \\ \log P}}{=} \frac{1}{\frac{\log N - \log P}{\log N} + \frac{c_1 + c_3}{c_2} \cdot \frac{\log^2 P}{\log N}} = \\ &= \frac{1}{1 - \frac{\log P}{\log N} + \frac{c_1 + c_3}{c_2} \cdot \frac{\log^2 P}{\log N}} \end{aligned}$$

- Eine isoeffiziente Skalierung erfordert somit

$$\frac{\log^2 P}{\log N} = K$$

und somit $N(P) = \Theta(P^{\log P})$,da $\log N(P) = \log^2 P \iff N(P) = 2^{\log^2 P} = P^{\log P}$). Wegen $W = \Theta(N \log N)$ folgt aus $N \log N = P^{\log P} \cdot \log^2 P$ für die Isoeffizienzfunktion

$$W(P) = \Theta(P^{\log P} \log^2 P).$$

- Der Algorithmus ist also *sehr* schlecht skalierbar!!

17.4 Paralleles Quicksort

- Quicksort besteht aus der Partitionierungsphase und dem rekursiven Sortieren der beiden Teilmengen.
- Naiv könnte man versuchen, immer mehr Prozessoren zu verwenden, je mehr unabhängige Teilprobleme man hat.
- Dies ist nicht kostenoptimal, da die Teilprobleme immer kleiner werden.
- Formal erhält man:

$$\begin{aligned} T_S(N) &= \underbrace{N + 2\frac{N}{2} + 4\frac{N}{4} + \ldots}_{\substack{\log N \\ \text{Schritte} \\ \text{(average)}}} = \\ &= N \log N \\ T_P(N) &\underset{\substack{N = P \\ \text{naive} \\ \text{Variante}}}{=} N + \frac{N}{2} + \frac{N}{4} + \cdots = \\ &= 2N \end{aligned}$$

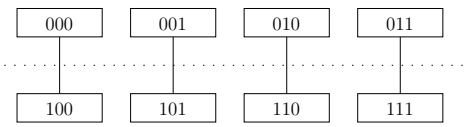
- Für die Kosten erhält man ($P = N$)

$$P \cdot T_P(N) = 2N^2 > N \log N$$

also nicht optimal.

Fazit: Man muss den Partitionierungsschritt parallelisieren, und das geht so:

- Es sei $P = 2^d$, jeder hat N/P Zahlen.
- Wir benutzen die Tatsache, dass ein Hypercube der Dimension d in zwei der Dimension $(d-1)$ zerfällt, und jeder im ersten Hypercube hat genau einen Nachbarn im zweiten:



- Nun wird das Pivot x gewählt und an alle verteilt, dann tauschen die Partnerprozessoren ihre Blöcke aus.
- Die Prozessoren im ersten Hypercube behalten alle Elemente $\leq x$, die im zweiten Hypercube alle Elemente $> x$.
- Dies macht man rekursiv d mal und sortiert dann lokal in jedem Prozessor mit Quicksort.
- Komplexität (average case: jeder behält immer $\frac{N}{P}$ Zahlen):

$$T_P(N,P) = \underbrace{c_1 \operatorname{ld} P \cdot \frac{N}{P}}_{\text{split}} + \underbrace{c_2 \operatorname{ld} P \cdot \frac{N}{P}}_{\text{Komm.}} + \underbrace{c_3 \frac{N}{P} \operatorname{ld} \frac{N}{P}}_{\substack{\text{loka.}\\ \text{Quicksort}}} + \underbrace{c_4 \log^2 P}_{\substack{\text{Pivot}\\ \text{Broad-}\\ \text{cast}}};$$

- Der Aufwand für den Broadcast des Pivot ist $\operatorname{ld} P + \operatorname{ld} \frac{P}{2} + \cdots + \operatorname{ld} 2 = d + (d-1) + (d-2) + \cdots + 1 = \sum_{i=1}^{d} i = \frac{d\cdot(d-1)}{2} = O(d^2)$.
- Für die Effizienz erhält man

$$\begin{aligned} E(N,P) &= \frac{c_3 N \operatorname{ld} N}{\left(c_3 \frac{N}{P} \operatorname{ld} \frac{N}{P} + (c_1+c_2)\frac{N}{P} \operatorname{ld} P + c_4 \operatorname{ld}^2 P\right) P} = \\ &= \frac{1}{\frac{\operatorname{ld} N - \operatorname{ld} P}{\operatorname{ld} N} + \frac{c_1+c_2}{c_3} \cdot \frac{\operatorname{ld} P}{\operatorname{ld} N} + \frac{c_4}{c_3} \cdot \frac{P \operatorname{ld}^2 P}{N \operatorname{ld} N}} = \\ &= \frac{1}{1 + \left(\frac{c_1+c_2}{c_4} - 1\right) \frac{\operatorname{ld} P}{\operatorname{ld} N} + \frac{c_4}{c_3} \cdot \frac{P \operatorname{ld}^2 P}{N \operatorname{ld} N}}. \end{aligned}$$

- Für eine isoeffiziente Skalierung ist der Term aus dem Broadcast entscheidend:

  $$\frac{P \operatorname{ld}^2 P}{N \operatorname{ld} N} = K,$$

  für $N = P \operatorname{ld} P$ erhalten wir

$$\begin{aligned} N \operatorname{ld} N &= (P \operatorname{ld} P)\ \operatorname{ld}(P \operatorname{ld} P) = (P \operatorname{ld} P)\ (\operatorname{ld} P + \underbrace{\operatorname{ld} \operatorname{ld} P}_{\substack{\text{sehr}\\ \text{klein!}}}) \approx \\ &\approx P \operatorname{ld}^2 P. \end{aligned}$$

- Mit einem Wachstum $N = \Theta(P \operatorname{ld} P)$ sind wir also auf der sicheren Seite (es genügt sogar etwas weniger).
- Für die Isoeffizienzfunktion gilt wegen $W = N \log N$

  $$W(P) = \Theta(P \operatorname{ld}^2 P),$$

  also deutlich besser als bei bitonischem Sortieren.

Empfindlichkeit gegenüber Pivotwahl

- Das Problem dieses Algorithmus ist, dass die Wahl eines schlechten Pivotelements zu einer schlechten Lastverteilung in *allen* nachfolgenden Schritten führt und somit zu schlechtem Speedup.
- Wird das allererste Pivot maximal schlecht gewählt, so landen alle Zahlen in *einer* Hälfte und der Speedup beträgt höchstens noch $P/2$.
- Bei gleichmäßiger Verteilung der zu sortierenden Elemente kann man den Mittelwert der Elemente eines Prozessors als Pivot wählen.