technische universität dortmund

Crashkurs Wissenschaftliches Rechnen in der Praxis: Parallele Numerik und ihre Umsetzung

Dominik Göddeke
dominik.goeddeke@math.tu-dortmund.de

Vorlesung Wissenschaftliches Rechnen
Fakultät für Mathematik
15. Und 23. Juni 2010

Kontext dieser Vorlesung

- Bisher:
  - „Hochleistungs-Numerik“: Finite Elemente Diskretisierungen und Mehrgitterlöser, theoretischer Hintergrund, Konvergenzbeweise
- Wissenschaftliches Rechnen (Scientific Computing) ist aber mehr
  - Übertragung auf praxisrelevante Probleme
  - Implementierung guter Numerik auf konkreter Hardware
  - Sehr schnell sehr interdisziplinär: Modellierung, Numerik, Informatik, …
  - High Performance Computing
- Diese Vorlesung (und die am 29.6.)
  - Crashkurs **Parallele** Numerik und insbesondere ihre technische Umsetzung auf konkreter Hardware
  - Ziel: Grundlagen über den Tellerrand hinaus, Überblick schaffen
  - Kein Ziel: Programmierkurs
  - Folien bald im Netz: http://www.mathematik.tu-dortmund.de/~goeddeke/temp/VL_WissRech.pdf

- Ein Schwerpunkt unserer Forschung
  - Problematik: Gute Numerik passt (nicht mehr) zur Hardware
  - Schlaue Implementierung alleine reicht nicht um ewig lange Rechenzeiten zu vermeiden
- Unser Ziel
  - Entwicklung numerischer Algorithmen die gut zur Hardware passen, jetzt und zukunftssicher
- Mehr dazu morgen in der Vorlesung von Hilmar Wobker
  - Heute und am 29.6.: notwendige Grundlagen und Konzepte
  - Rest der Vorlesungszeit: konkrete Beispiele und Vertiefung anhand aktueller Forschungsergebnisse
- Einladung
  - Seminararbeiten, Bachelor- und Masterarbeiten (bei uns idR Theorie+Umsetzung) in diesem Kontext sind immer möglich

Inhalt dieser Vorlesung

- Einführung
  - Begriffsbildungen
  - Beispiel: Crash-Simulation (zur Verdeutlichung der grundlegensten Dinge)
  - Memory Wall Problematik
- Theoretischer Hintergrund und parallele Modellierung
  - Speedup, Skalierbarkeit, Amdahl
- Parallelisierungstechniken
  - Gebietszerlegung, funktionale Zerlegung, Lastverteilung
- Rechnerarchitekturen
  - Klassifizierung nach Flynn, shared vs. distributed memory
- Parallele Programmiermodelle
  - Prozesse und Threads
  - MPI und OpenMP für Einsteiger

- Dank an Bernd Mohr (JSC) für die „Inspiration“ für weite Teile dieser Folien

Einführung und Begriffsbildung

Warum Parallelisierung

- Parallele Computer können die einzige Möglichkeit sein, Probleme im wissenschaftlichen Rechnen in akzeptabler Zeit zu lösen
  - Serielle Computer („mein Laptop mit Matlab“) sind zu langsam
  - Rechnung dauert Stunden, Tage, Wochen, Monate
  - Verwende mehr als einen Computer um die Antwort schneller zu erhalten (später: starke Skalierbarkeit)
- Parallele Computer können die einzige Möglichkeit sein, große praxisrelevante Probleme zu lösen
  - 4 GB Speicher in meinem Laptop sind zu wenig
  - Parallele Systeme haben mehr Speicher pro Computer (später: schwache Skalierbarkeit)
- Auch immer wichtiger:
  - Hardware wird parallel (Multicore (mein Laptop), GPUs, …) und wir wollen das ausnutzen

Begrifflichkeiten

- Paralleles Rechnen (parallel computing)
  - Berechnung eines gegebenen Problems durch Verwendung vieler Prozessoren bzw. Rechenressourcen
  - Homogen: Alle Prozessoren sind gleich (aus der Sicht des Systems, d.h. Rechnung *und* Kommunikation zwischen ihnen)
  - Heterogen: Prozessoren unterscheiden sich
    - Speicheranbindung, Caches, Kommunikation
    - Architektur: CPUs und GPUs
- Trivial parallele Probleme (embarrassingly parallel)
  - Alle Teilprobleme sind komplett unabhängig
  - Parameterstudien (heißt auch „farming“)
  - Langweilig!
- High Performance Computing (HPC)
  - Schnelle und große Supercomputer nutzen und ausnutzen (das ist nicht dasselbe!) für große und rechenintensive Probleme

Instruktives Beispiel

- Crash-Test in der Automobilindustrie

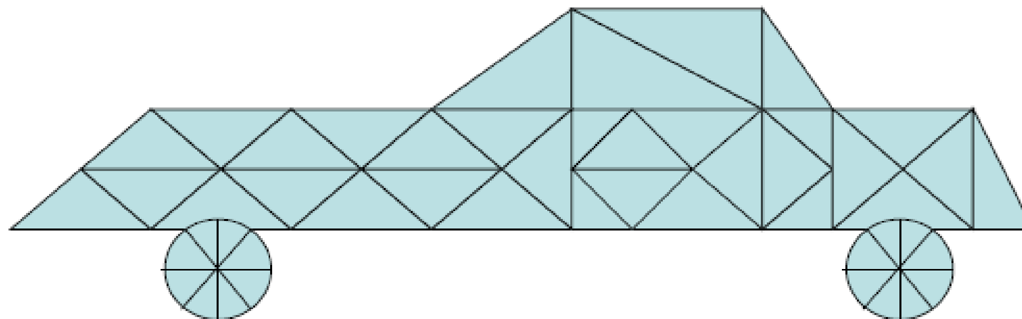
- Extremst stark vereinfachtes Beispiel
  - Tatsächliches Problem braucht viel mehr Mathematik und Multiphysik
  - Verschiedene Modelle aus der Strukturmechanik …

- Simulationen sparen Zeit und Geld
  - Echte Kfz-Prototypen bauen und zu verwenden ist teuer und gefährlich
  - Man will Kenntnisse schon in frühen Planungsphasen erhalten

- Ziel dieses Beispiels
  - Instruktive Darstellung wichtiger Grundsätze im parallelen Rechnen
  - All das lässt sich beliebig übertragen

FEM Diskretisierung

- Fahrzeug modelliert durch (feines) Netz von Elementen
  - Physikalische Größen nur in Eckpunkten des Gitters interessant
- Erster Schritt: Diskretisierung in der Zeit
- In jedem Zeitschritt
  - Änderung der Position der Elemente durch die einwirkenden Kräfte
  - Krafteinwirkung ist typischerweise ein lokaler Prozess, d.h., die Diskretisierung reduziert das Problem auf Wechselwirkung benachbarter Elemente untereinander



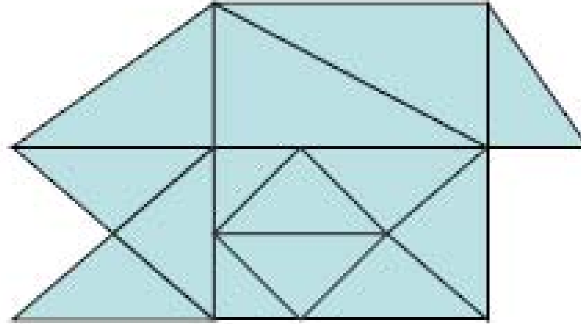
Pseudocode, seriell

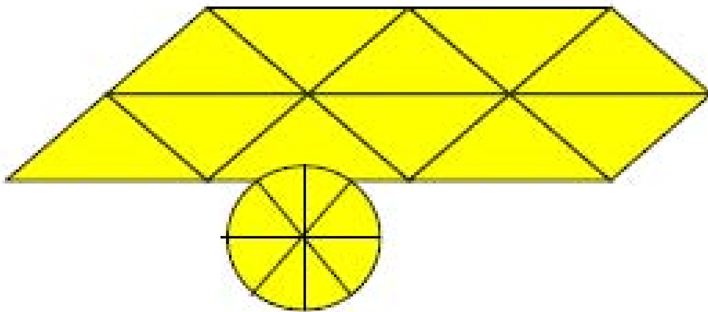
- Preprocessing: Für alle Elemente
  - Status, Eigenschaften, Nachbarliste einlesen
- Für jeden Zeitschritt
  - Für jedes Element
    - Status berechnen als Eingabe für den nächsten Zeitschritt

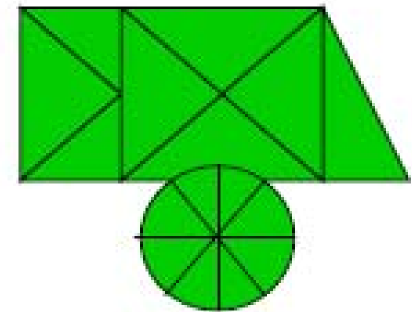
Einfacher Parallelisierungs-Zugang

- Cluster aus PCs
  - Verbunden mit schnellem Netzwerk (**distributed memory computer**)
  - Prozessoren kommunizieren über Nachrichten (**message passing**)
- Parallelisierung im Ort
  - Zeitschleife nicht parallelisierbar (inhärent sequentieller Charakter)
- Daten auf Prozessoren verteilen (**data distribution**)
  - Jeder Prozessor aktualisiert die Elemente, die in seinem Speicher liegen (**owner computes**)
- Alle Maschinen führen dasselbe Programm aus (**SPMD**)

Ein verteiltes Auto







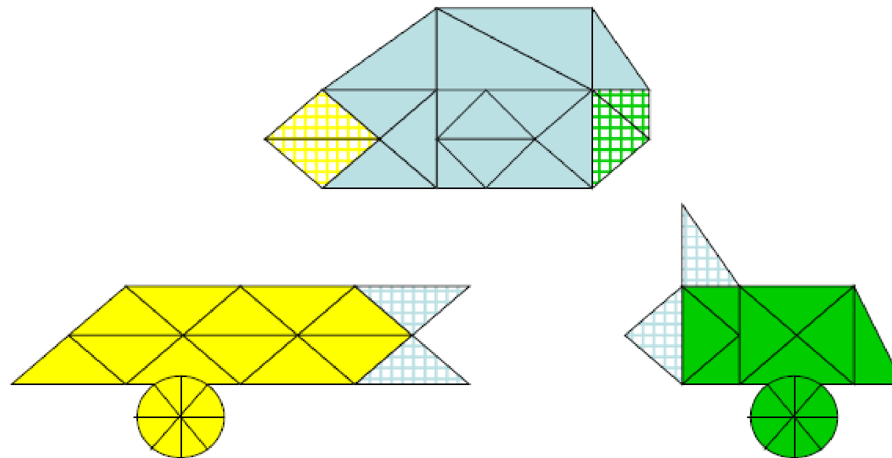
Pseudocode, parallel

- Gleichzeitig für alle p Prozessoren
- Preprocessing: Für alle Elemente auf Prozessor p
  - Status, Eigenschaften, Nachbarliste einlesen
- Für jeden Zeitschritt
  - Für jedes Element auf Prozessor p
    - Status berechnen als Eingabe für den nächsten Zeitschritt
    - Statusinformationen zwischen den benachbarten Elementen austauschen und synchronisieren, die auf verschiedenen Prozessoren liegen

Wichtige Aspekte

- Allocation: Wie werden Elemente den Prozessoren zugeordnet?
  - (Initiale) Zuordnung durch serielles Preprocessing mittels Gebietszerlegung (**domain decomposition**)
  - Oft dynamische Umverteilung nötig (**load balancing**)

- Gebietszerlegung und Kommunikation: Wie behalten die Prozessoren den Überblick, welche Elemente „geteilt sind"?
  - Überlappende Gebietszerlegung (**ghost cells, halo**)



Noch mehr wichtige Aspekte

- Update: Wie erhält ein Prozessor Informationen von benachbarten Elementen?
  - Anfrage schicken?
  - Effizienter: Prozessor schickt Daten weg, sobald sie berechnet sind
- Programmierung und Korrektheit: Software Engineering für parallele Programme?
  - Um Längen schwerer zu debuggen als serielle Programme
  - Verschiedene Programmierparadigmen (->später)
- Effizienz: Wie messen wir den Erfolg der Parallelisierung?
  - Speedup und parallele Effizienz (->später)

Der wichtigste Aspekt

- Für PDE-Simulationen auf Gittern
  - Nichts anderes interessiert uns ☺
- Memory Wall Problematik
  - Daten zu lesen und zu schreiben ist viel teurer als Rechnungen auf ihnen durchzuführen
  - Abstand zwischen Rechengeschwindigkeit und Speicherbandbreite wird immer größer
  - Rechnen: 60% schneller pro Jahr
  - Speichertransfers: 10% schneller pro Jahr
- Caches sind (in Verbindung mit reiner Compileroptimierung) keine Lösung
  - Unsere working sets sind zu groß, und wir nutzen Daten zu selten mehrmals

Beispiel (vereinfacht aber instruktiv)

- Addition zweier Vektoren der Länge n (zu lang für den Cache)
  - Rechenoperationen: n, Speicheroperationen: 3n, 8 Byte pro Wert
  - **Arithmetische Intensität**: 1/3
  - Mein Rechner: 12 GFLOP/s Rechnung, 10 GB/s Speicherbandbreite
- Um 12 GFLOP/s zu sehen, bräuchten wir (1/3)^-1 * 8 * 10 GB/s = 0.24 TB/s Speicherbandbreite
  - D.h. 24 mal so viel wie wir haben
  - Vergleich: 2/3 der Festplatte in meinem Laptop pro Sekunde kopieren
- Andersherum argumentiert
  - Wir können maximal (Sonntags, bei gutem Wetter und bergab) nur 12/24 = 0.5 GFLOP/s für diese Operation erreichen!
  - D.h. 4.2% der theoretischen Maximalleistung
- Moral: Betrachte die „richtige“ Kenngröße!

Evaluierung paralleler Programme

- Extrem wichtig!
  - Nachdem Korrektheit sichergestellt ist und alle deadlocks usw. eliminiert worden sind
- Problem: Programm läuft nicht effizient oder kann nicht nicht auf die gewünschte Zahl Prozessoren hochskaliert werden
  - Ursache finden: **Performance-Analyse**
  - Problem beheben:
    - Bessere Zugänge und Parallelisierungskonzepte entwickeln
    - Tuning und Optimierung
  - Im Gegensatz zur Analyse „schwer“ und Handarbeit (bzw. Kopfarbeit)

Theoretischer Hintergrund und parallele Modellierung

Definitionen

- Für ein gegebenes Problem A
  - **Ts(n)** = Laufzeit (asymptotisch) des besten bekannten seriellen Algorithmus zur Lösung von A bei Problemgröße n
  - **Tp(n,p)** = Laufzeit eines parallelen Algorithmus um A bei Problemgröße n mit p Prozessoren zu lösen
  - Wichtig: Ts(n) ≤ Tp(n,1), denn sonst wäre der parallele Algorithmus ausgeführt auf einem Prozessor ein besserer serieller, Widerspruch zur Annahme bestmöglicher serieller Algorithmus

- Damit
  - **Overhead To = p*Tp – Ts** (Zeit, die alle zusammen länger brauchen als der beste serielle Algorithmus)
  - **Speedup S = Ts / Tp**
  - **Kosten C = Tp * p** (Kosten des seriellen Algorithmus ist gerade Ts)
  - **Effizienz E = S / p = Ts / C = Ts / (p*Tp)**

Definitionen

- Zusammenhänge
  - $0 \leq S \leq p$
  - C(seriell) ≤ C(parallel) ≤ \Infty
  - $0 \leq E \leq 1$ (deshalb wird parallele Effizienz oft als Prozentsatz angegeben)

- Speedup
  - Perfekter oder idealer Speedup: $S(p) = p$
  - Superlinearer Speedup: $S(p) > p$ (bzw. $E(p) > 1$)
    - Kann passieren: Paralleles System kann mehr Daten im Cache halten (weil p mal so viele Caches zur Verfügung stehen)
    - Ist also kein Widerspruch zu Ts optimal

- Kosten
  - Paralleler Algorithmus heißt kostenoptimal, wenn $E=O(1)$

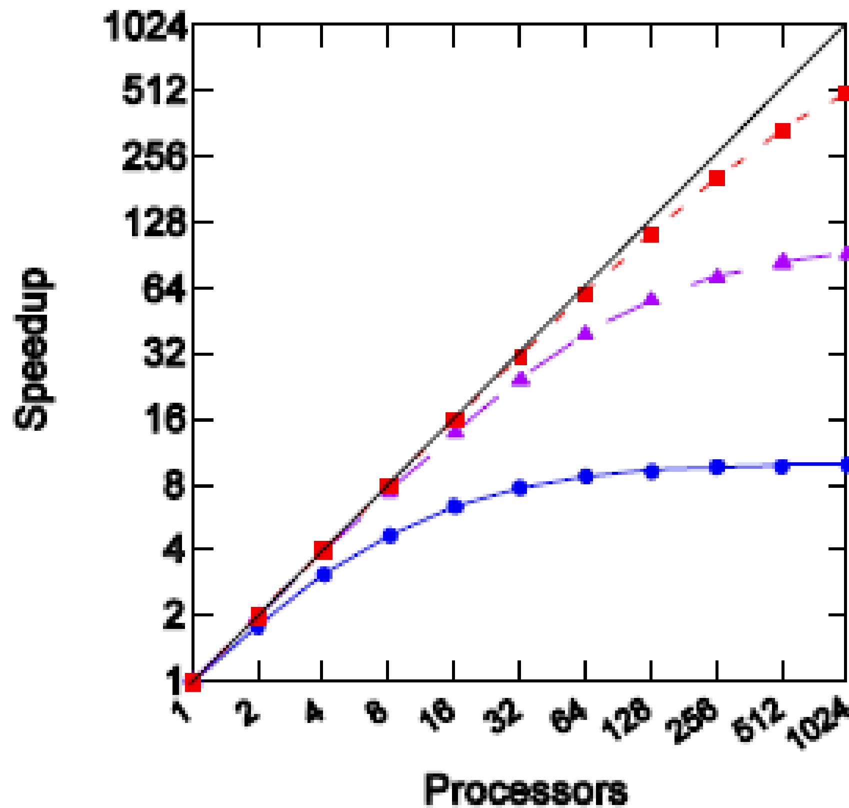
Beispiel

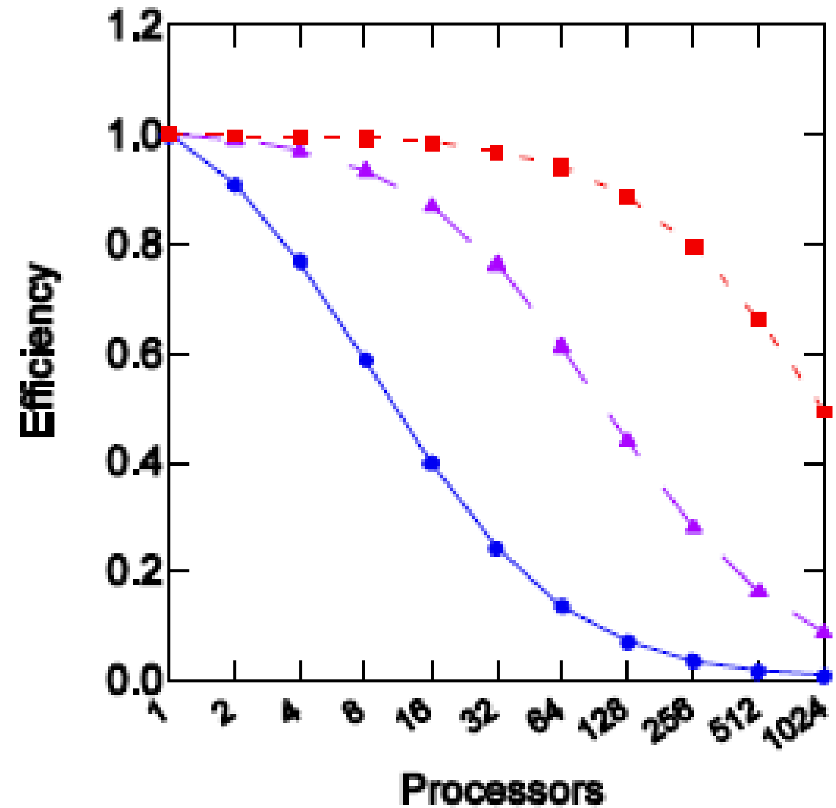
- N Zahlen mit n Prozessoren addieren, $n=2^k$
  - Serieller Algorithmus: klar, $T_s = \Theta(n)$
  - Paralleler Algorithmus: Binärbaum, $T_p = \Theta(\log n)$
- Speedup, Effizienz und Kosten
  - $S = \Theta(n/\log n)$
  - $E = \Theta(1/\log n)$
  - $C = \Theta(n * \log n)$
- Also ist dieser Algorithmus nicht kostenoptimal für p=n Prozessoren
  - Für $p < n$ Prozessoren kann dieser Algorithmus mit leichten Modifikationen kostenoptimal gemacht werden
  - „Übungsaufgabe“ ☺

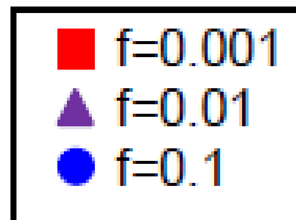
Amdahl‘s Law

- Amdahl (1967)
  - Gegeben ein Programm A. f bezeichne den Anteil an Operationen, die nicht parallelisierbar sind und somit sequentiell ausgeführt werden müssen. Dann gilt für p Prozessoren
    $$S \leq 1 / (f + (1-f)/p)$$
  - Unter der Annahme: Paralleler Teil (1-f) Faktor p schneller (idealer Speedup)
- Konsequenz für beliebig viele Prozessoren (p -> \Infty)
  - $S \leq 1/f$
- Schlechte Nachricht
  - In typischen Programmen ist f nicht vernachlässigbar

Maximaler Speedup / Effizienz







Amdahl‘s Law

- Amdahl war ein Optimist
  - Parallelisierung erfordert oftmals mehr Arbeit
  - Amdahl bedeutet dann: Parallelisieren macht keinen Sinn, weil es sich sowieso nicht lohnt.
  - Doof bzw. keine wissenschaftlich akzeptable Konsequenz
- Amdahl war ein Pessimist (besser für uns)
  - Konstruiere parallele Algorithmen mit minimalem seriellen Anteil f
  - Nutze superlinearen Speedup aus (d.h. konstruiere Algorithmen die das können indem sie besser zur Hardware passen)
  - Oder, generell und prinzipiell schlauer, **Skalierbarkeit**: Zeit, die in f verbracht wird, ist oft nicht proportional zur Gesamtlaufzeit, wenn die Gesamtproblemgröße hinreichend erhöht wird

Skalierbarkeit

- In der Praxis: Eigenschaften von f
  - Benötigt konstante Zeit unabhängig von der Problemgröße n
  - Oder wächst asymptotisch langsamer als die Gesamtlaufzeit (so dass der parallele Anteil skalierbar lange dominiert)

- Also können parallele Maschinen effizient genutzt werden
  - Wenn die Problemgröße mit der Zahl der Prozessoren skaliert wird

- Zwei Zugänge in der Praxis, vier in der Theorie
  - Problemgröße konstant, p hoch (**strong scaling**)
  - Problemgröße pro Prozessor konstant, p hoch (**weak scaling**)
  - Konstante Zeit, finde größte skalierbar effizient lösbare Problemgröße (**Gustafson‘s Law**)
  - Effizienz konstant, wie klein darf das Problem maximal werden im Verhältnis zu p (**Isoeffizienz-Analyse**)

Parallelisierungstechniken und Lastverteilung

Parallelisierungstechniken

- Ziel: Arbeit und/oder Kommunikation geeignet zwischen Prozessoren aufteilen

- Gebietszerlegung (domain decomposition)
  - Partitionierung eines (evtl. nur konzeptionellen) Raumes
  - Bei uns üblicherweise das Gebiet im klassischen PDE-Sinne
  - Verschiedene Prozessoren führen (fast) dieselben Operationen auf verschiedenen Teilgebieten aus
  - Crash-Simulation von eben

- Funktionale Zerlegung
  - Verschiedene Prozessoren arbeiten verschiedene Aufgaben ab
  - Beispiel: Fliessband: Daten wandern von Prozessor zu Prozessor
  - Oft zu teuer / kommunikationsintensiv

Parallelisierungstechniken

- Datenparallele Rechnung
- Task-Graph
- Work Pool (Arbeiter holen sich an zentraler Stelle Arbeit ab)
- Master-Worker (Master verteilt Arbeit explizit an Slaves)
- Pipeline bzw. Producer-Consumer

Lastverteilung

- Idealistisches Ziel
  - Arbeit und Kommunikation perfekt gleichverteilen
- Probleme:
  - Hardware ist uU inhomogen
  - Nicht alle Teilgebiete erfordern den gleichen Rechenaufwand (Beispiel: irgendwo ist eine fiese Nichtlinearität versteckt und ein Teilgebiet konvergiert viel langsamer)
  - Dynamische Prozesse, adaptive Verfahren
- Lastverteilungs-Ansätze
  - Statisch a priori oder dynamisch
  - Parametrisiert oder datenabhängig
  - Homogen oder inhomogen (mehrstufige Lastverteilung)
  - Ein- oder mehrdimensionaler Optimierungsraum (Lastverteilung ist iA ein NP-hartes Problem!)

Rechnerarchitekturen

Taxonomie von Rechnerarchitekturen

- Klassifizierung erfolgt zwar auf der Basis von Hardware
  - Aber Software ist vollkommen analog
- Sinn und Zweck dieser Klassifizierungen
  - Mentales Modell, um über Probleme und ihre Lösung im Sinne von abstrakten Maschinenmodellen auf hoher und detaillierter Ebene nachdenken zu können und Algorithmen geschickt zu entwerfen
- Echte Hardware
  - Oft (immer) eine Mischung dieser Charakterisierungen

Flynn‘s Taxonomie

technische universität dortmund

- Flynn 1966: Auftrennung nach Instruktionen und Daten

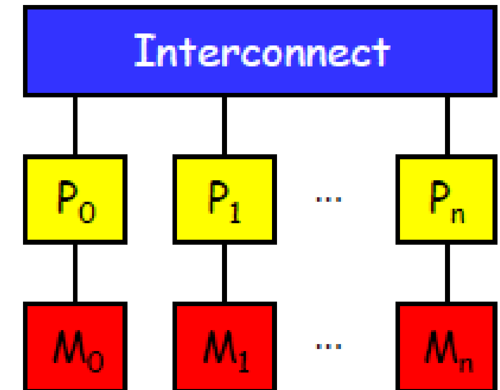
- SI: Single Instruction
  - Alle Prozessoren führen dieselbe Instruktion aus
- MI: Multiple Instruction
  - Verschiedene Prozessoren dürfen verschiedene Operationen ausführen
- SD: Single data
  - Alle Prozessoren arbeiten auf denselben Daten
- MD: Multiple data
  - Alle Prozessoren arbeiten auf denselben Daten

Flynn‘s Taxonomie

- SISD
  - „Standard“ serieller Computer bzw. serielles Programm
- MISD
  - Sehr selten, fehlertolerante Systeme über Redundanz (Space Shuttle)
- MIMD
  - Fast alle parallelen Supercomputer (und mittlerweile auch mein Laptop)
- SIMD
  - Frühere Supercomputer (Vektorrechner)
  - Short-SIMD in Standardprozessoren (SSE)
  - GPUs (naja, fast)

Distributed Memory

- Über ein Netzwerk verbundene Rechenknoten
  - Knoten = Prozessor und Speicher
  - Alle zusammen: Cluster



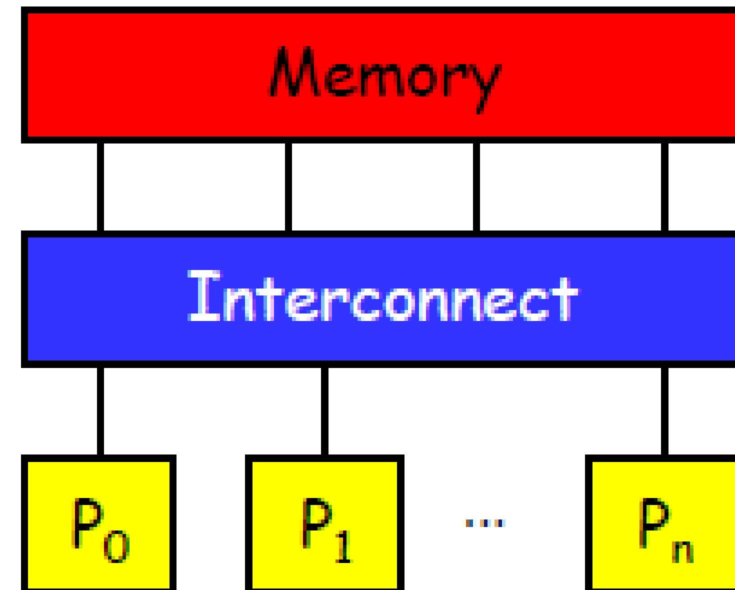
- Speicher gehört lokal zu einem Prozessor

- Vorteile
  - Speicher ist mit Zahl der Prozessoren skalierbar
  - GROSSE Machinen möglich mit > 10000 Knoten (modulo Energiebedarf)
  - Jeder Prozessor kann seinen Speicher einfach und ohne Störung von außen ansprechen
  - Kosteneffizient und einfach mit Standardkomponenten konstruierbar

- Nachteile
  - Um Informationen von einem anderen Prozessor zu erhalten, müssen explizit Nachrichten über das Netzwerk geschickt werden
  - Programmierer ist für fast alles selbst verantwortlich
    - Explizite Datenverteilung
    - Explizite Kommunikation
    - Explizite Synchronisierung
  - Oft nicht intuitiv, zusätzliche Datenstrukturen nötig
    - Beispiel Crash-Simulation: Ghost Cells
- Programmiermodell
  - Message Passing, MPI
  - Datenparallelität: HPF (eher ein Exot)
- Praxis
  - MPI ist heute de-facto Standard für ALLE Cluster

Shared Memory

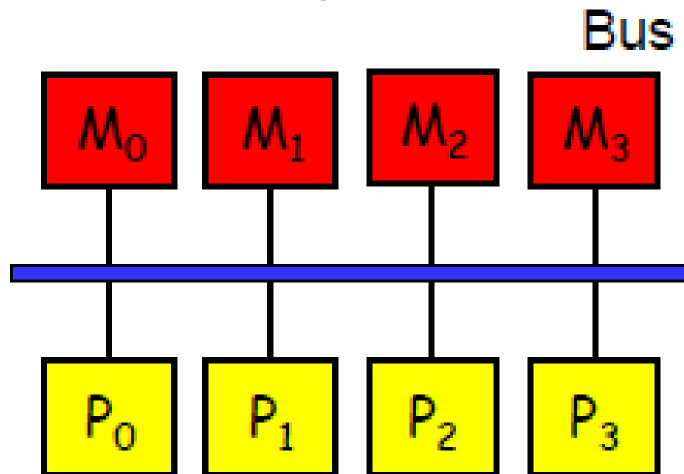
- Genauer: geteilter Addressbereich, auf den alle Prozessoren zugreifen können
- Prozessoren dürfen lokalen, privaten Speicher haben
  - Caches, Kohärenz wird idR durch die Hardware sichergestellt
- Programmiermodelle
  - Autoparallelisierung (Compiler)
  - Explizites threading (POSIX threads)
  - OpenMP

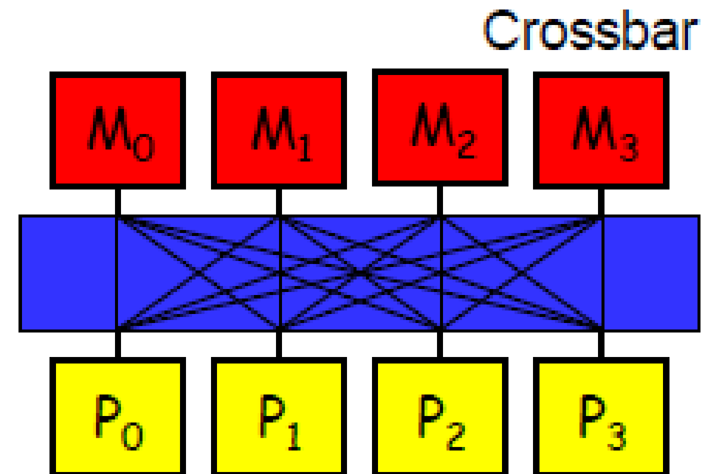


Shared Memory

- Vorteile
  - Serielle Datenstrukturen müssen kaum geändert werden
  - Die Hardware übernimmt, wenn man ihr das „geeignet“ mitteilt, viele Aufgaben um auf die Daten effizient zugreifen zu können
  - Globaler Addressbereich macht Programmierung einfacher
    - Implizite Kommunikation auf geteilten Daten (jeder kann immer alles lesen)
    - Explizite Synchronisation
  - Kommunikation ist um Größenordnungen schneller als im distributed memory Fall
- Nachteile
  - Nicht wirklich Hardware-seitig skalierbar
  - Wird irgendwann zu teuer

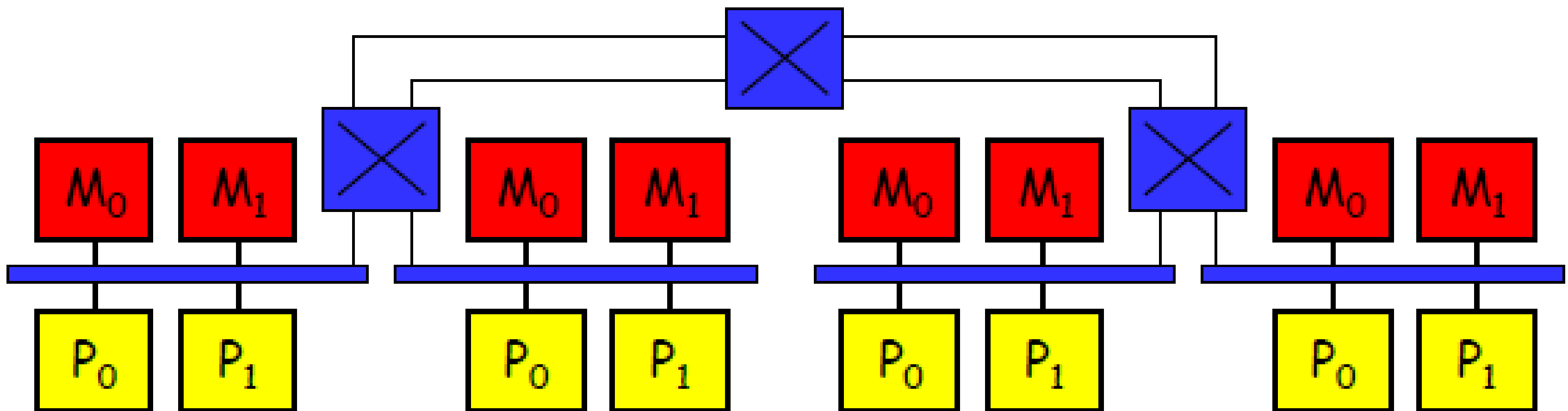
- Weitere Aufteilung gemäß Speicherzugriffszeit
- UMA (uniform memory access)
  - Gleiche Zugriffszeit, idR Cache-kohärent





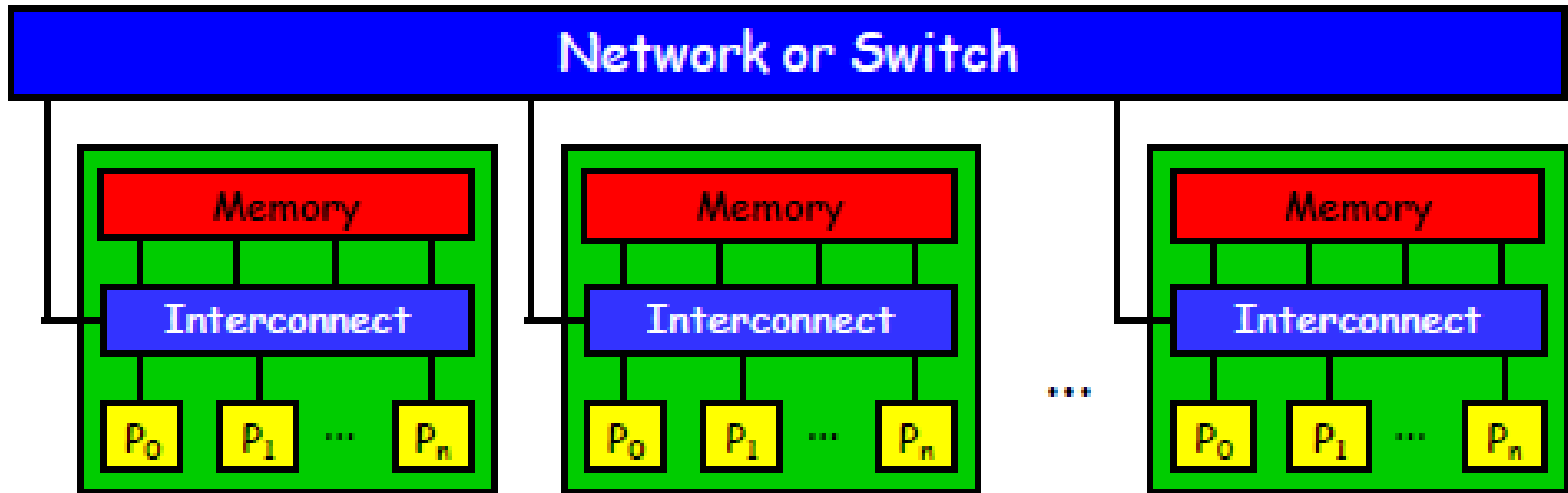
  - Gängige Praxis in heutigen Multicores
  - Kaum skalierbar
  - Andere Namen: SMP (symmetric multiprocessor), CMP (chip multi-processor)

- NUMA (non-uniform memory access)
  - UMA-Prozessoren durch Mainboard-Interconnect verbunden
  - Gängige Praxis in einem Clusterknoten (mehrere Multicores pro Knoten)
  - Auch: cc-NUMA,



Heute: Hybride Architekturen

- Heterogene, mehrstufige Speicherarchitektur
  - Programmiermodell unklar
  - MPI oder „MPI + X“ ☺
- Programmierung immer anspruchsvoller
  - Aber nötig, um nicht „immer langsamer“ pro Core zu werden



Praxis: Kommunikationskosten

- Kommunikationskosten in hybriden distributed memory Architekturen müssen berücksichtigt werden
  - Vielstufige Speicherhierarchie
  - Memory wall!
- Latenz und Bandbreite
  - Kommunikation zwischen Rechenknoten ist um Größenordnungen langsamer als Kommunikation innerhalb eines Rechenknotens
  - Cache-Zugriff ist um Größenordnungen schneller als Speicherzugriff
- Lokalität, Lokalität, Lokalität!
- Algorithmen müssen so entworfen UND implementiert werden, dass sie die Speicherhierarchie transparent ausnutzen können

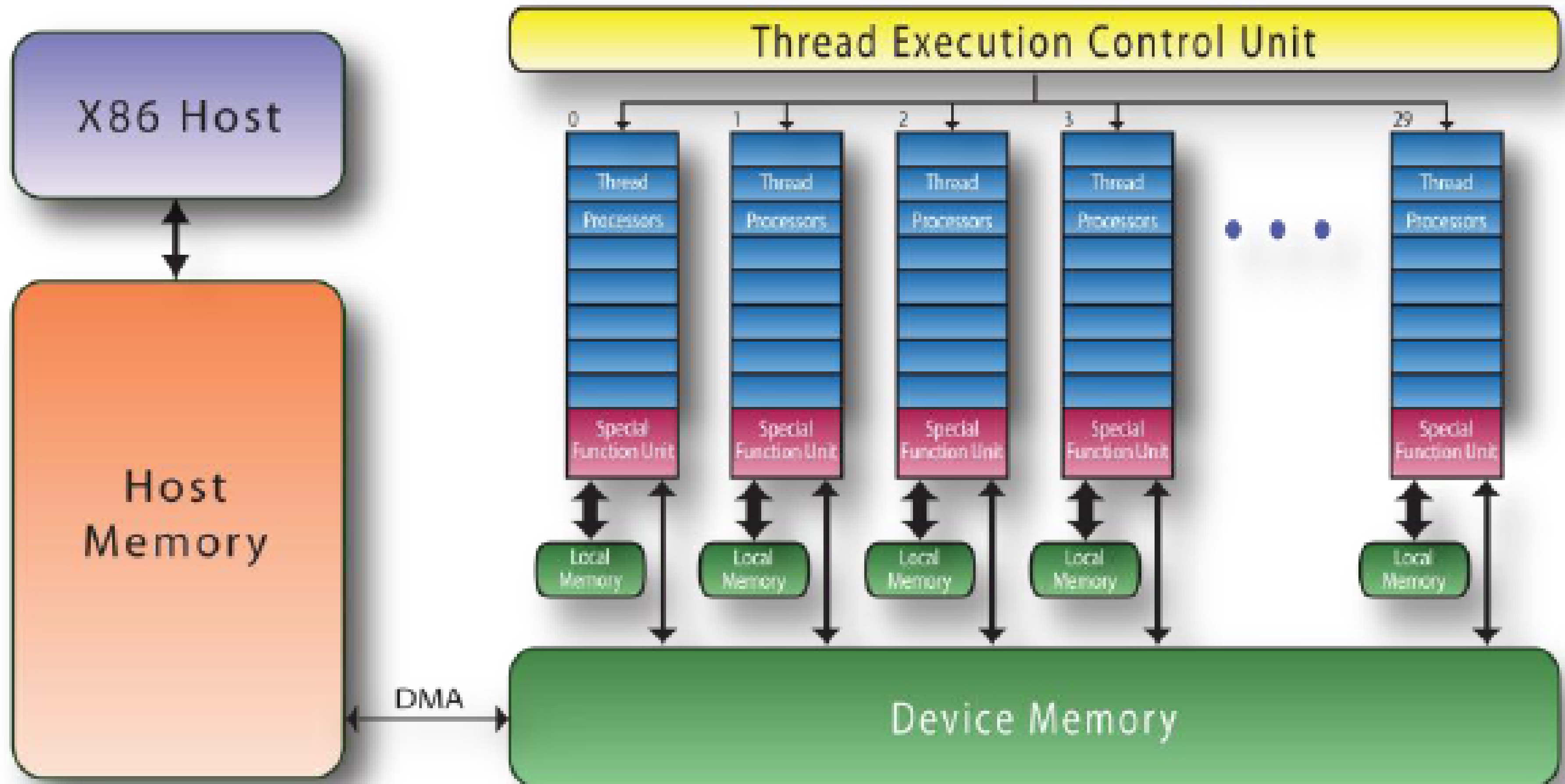
Praxis: Accelerators / Koprozessoren

- Zusatzhardware zur beschleunigten Lösung von Teilaufgaben
  - Lange Tradition in HPC-Rechnersystemen
  - Und „neuerdings“ auch in Standard-Hardware

- Beispiele
  - MMX, SSE, 3DNow! etc: Vektor-Operationen in CPUs (führe vier Additionen gleichzeitig aus, wenn die Daten „geeignet“ im Speicher liegen)
  - FPGAs (field programmable gate arrays): „Bau dir deinen eigenen special-purpose Prozessor“

- Sehr aktuell: GPUs (Grafikkarten)
  - Unterstützen bis 30000 Threads simultan, anderes Programmiermodell
  - Schnell! (und eigentlich nicht superschwer zu programmieren)
  - Mehr in einer späteren Vorlesung

Beispielknoten



Parallele Programmiermodelle

Parallelisierungstechniken (II)

- Rechenressourcen (wegen memory wall)
  - Prozessor
  - Speicher
- Parallelisierung bedeutet
  - Arbeit zwischen Prozessoren verteilen
  - Arbeit zwischen Prozessoren synchronisieren
  - Wenn auch der Speicher verteilt ist: Daten zwischen Speicherbereichen verteilen und Daten zwischen Speichern kommunizieren
- Programmiermodell
  - Kombinierte Methoden für Arbeits- und Datenverteilung
  - Kommunikation und Synchronisation

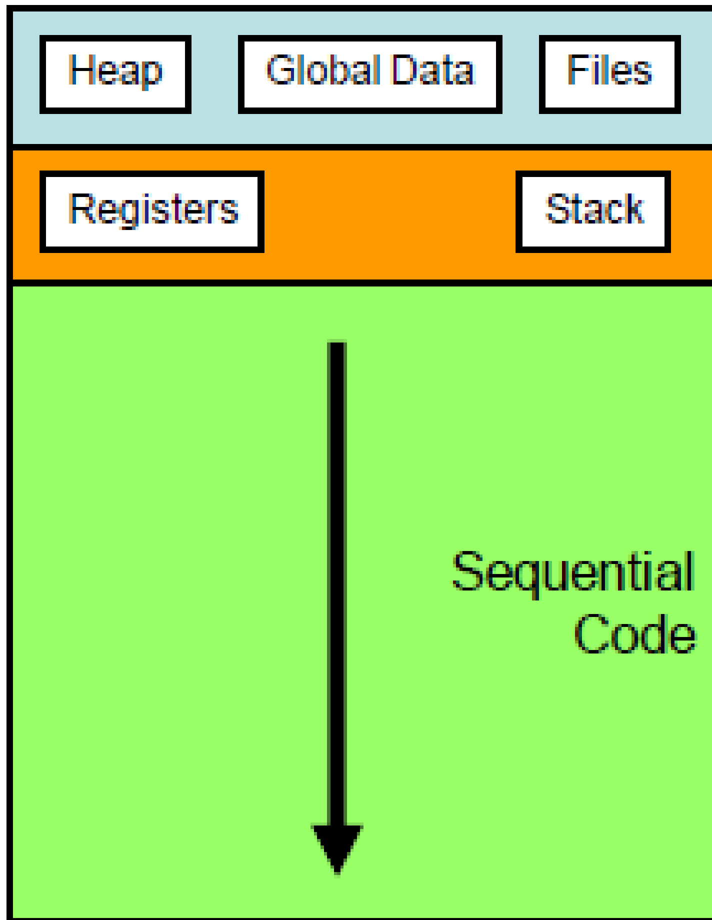
Prozesse und Threads

- Prozesse werden vom Betriebssystem bereitgestellt und führen Programme aus

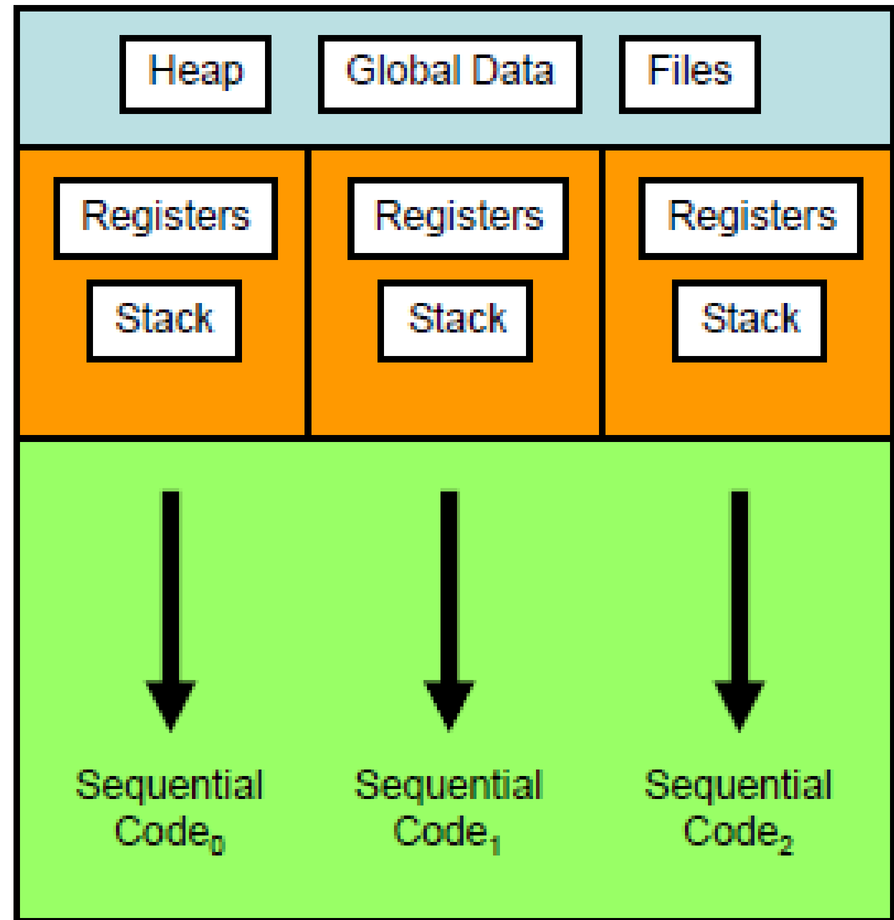
- Serieller Prozess = Thread of execution kombiniert mit
  - Stack und Heap (Speicherbereiche)
  - Instruktionszeiger

- Multithreading
  - Ein Prozess kann mehrere Threads beinhalten (dynamisch generiert und zerstört zur Laufzeit)
  - Threads haben Zugriff auf den Speicher des gesamten Prozesses
  - Jeder Thread hat seinen eigenen Stack und Instruktionszeiger

Single-threaded vs. Multi-threaded



Single-threaded process



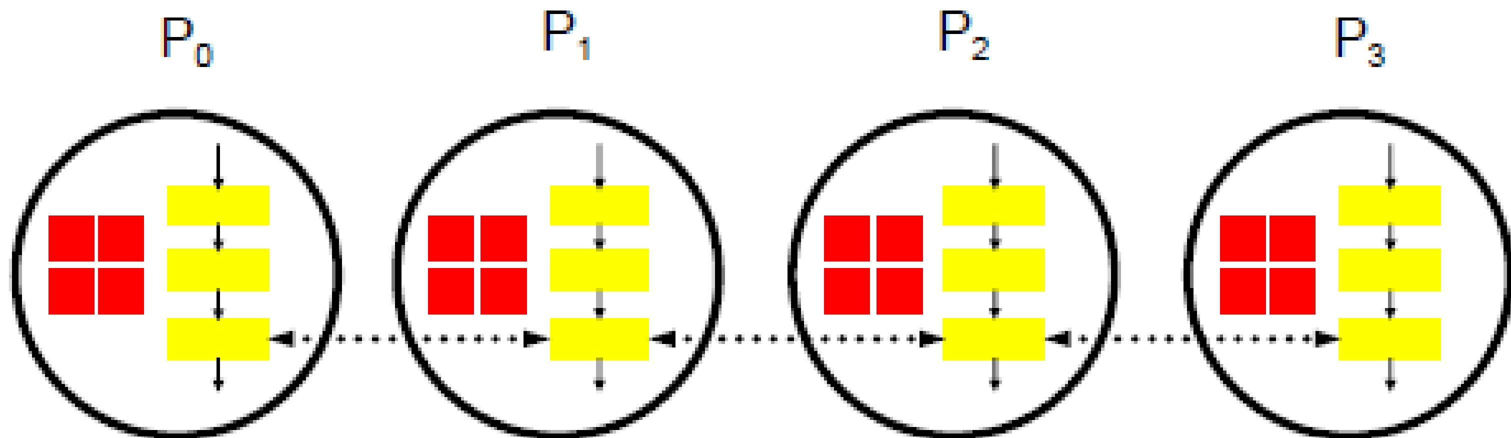
Multi-threaded process

Parallele Programmiermodelle: MPI für distributed memory

- Single Program Multiple Data (SPMD)
  - Programmierer schreibt EIN Programm
  - Programm wird auf ALLEN Prozessoren ausgeführt
  - Und zwar so, dass jeder Prozess auf UNTERSCHIEDLICHEN Teilen der Daten arbeitet
  - Divergenz und Konvergenz des Programmflusses werden im Programm selbst behandelt

```
if (process_or_thread_id == 42) then
  call do_something()
else
  call do_something_else()
endif
```

- Distributed Memory Architekturen
- SPMD-Programm arbeitet auf lokalen Daten
  - Explizite Gebietszerlegung, Datenverteilung, Kommunikation und Synchronisation
- MPI: De-facto Standard
  - www.mpi-forum.org, www.open-mpi.org



Message Passing Performance

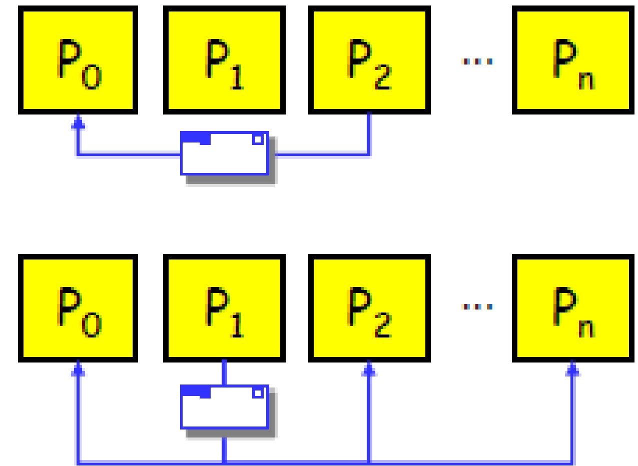
- Latenz
  - Kosten, eine Nachricht überhaupt loszuschicken
- Bandbreite
  - Kosten, viele Daten in einem Zeitintervall zu verschicken

- Latenzen sollten versteckt werden
  - Mehrere Nachrichten in einer zusammenfassen
  - Wenn P Daten hat die Q benötigt: Nicht von Q anfordern lassen, sondern P losschicken lassen sobald vorhanden. Bis Q die Daten braucht, ist die Nachricht hoffentlich schon angekommen
  - Send early, receive late, don‘t ask but tell ☺

- Idealfall: Kommunikation mit unabhängiger Berechnung überlappen
  - Geht nicht immer, wenn es geht, ist Kommunikation faktisch umsonst

MPI Funktionalität

- Point-to-point Kommunikation
  - zwischen zwei Prozessen
- Collective Kommunikation
  - In einer Gruppe von Prozessen
- All-to-one Kommunikation
  - Normberechnung, Skalarprodukt
- Barrier Synchronisation
  - Alle warten aufeinander, erst dann geht es weiter
- Einseitige Kommunikation, Parallel I/O, Process Creation etc.



MPI Communicators

technische universität dortmund

- Communicator
  - Gruppe von Prozessen zusammen mit gemeinsamen Kontext
  - Standard-Communicator für alle Prozesse: MPI_COMM_WORLD
- Versenden von Nachrichten erfolgt relativ zum aktuellen Communicator
  - Alle Prozesse in einem Communicator nehmen an collective Operationen teil
- Operationen / Unterstützung für
  - Zahl der Prozesse in einem Communicator
  - Rank des aktuellen Prozesses im Communicator
  - Prozesse in Unter-Communicatoren dynamisch zur Laufzeit aufteilen und umgrupppieren

Hello, World mit MPI


```
program main
include 'mpif.h'
integer :: ierr, myrank, numprocs

call MPI_Init(ierr)
call MPI_Comm_rank(MPI_COMM_WORLD, myrank, ierr)
call MPI_Comm_size(MPI_COMM_WORLD, numprocs, ierr)

write(*,*) "hello from", myrank, "of", numprocs

call MPI_Finalize(ierr)
end program
```

Fortran MPI routines:
error code returned in
extra parameter!

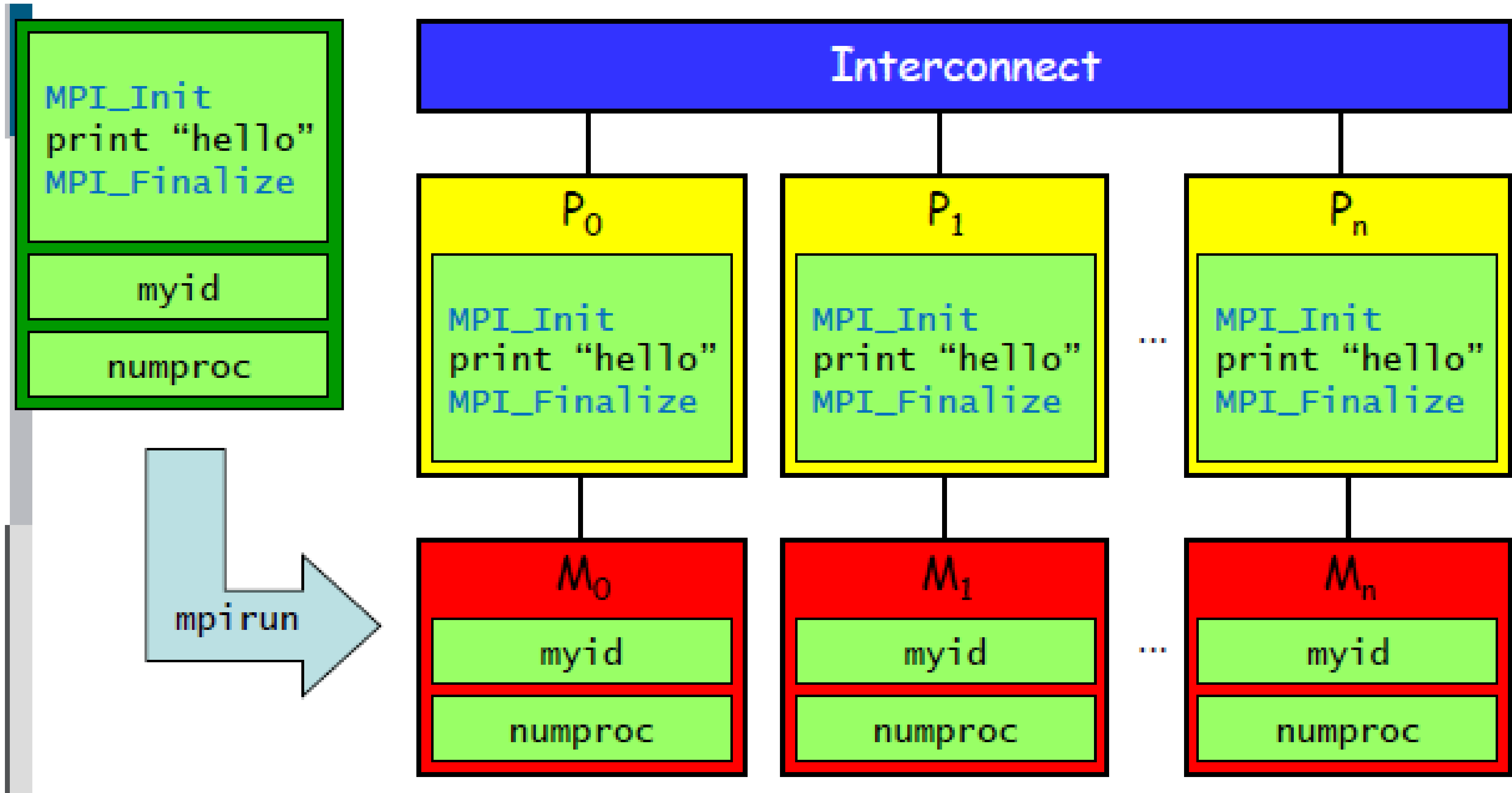
Hello, World mit MPI


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

int main(int argc, char *argv[])
{
  int ierr, myrank, numprocs;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);

  printf("hello from %d of %d\n", myrank, numprocs);

  MPI_Finalize();
}
```

C MPI_Init:
needs access to
program parameters

C:
error code returned
but ignored

MPI Programmablauf



MPI Programmablauf

- mpirun –np 4 helloworld

```
hello from 0 of 4
hello from 1 of 4
hello from 2 of 4
hello from 3 of 4
```

- mpirun –np 4 helloworld

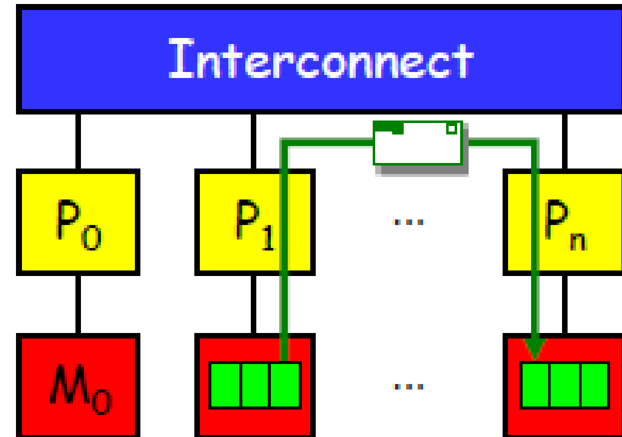
```
hello from 3 of 4
hello from 1 of 4
hello from 0 of 4
hello from 2 of 4
```

- mpirun –np 4 helloworld

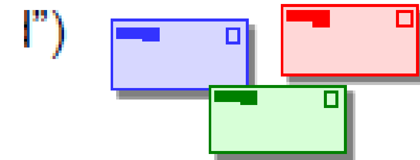
```
hehellhelllo from 3
lo from helf 4lo from
1 of 4o fr 2 of 4
om 0 of 4
```

MPI Nachrichten

- Nachricht =
  - buffer: Adresse der Daten
  - num: Anzahl der Daten
  - type: Datentyp
    - MPI_INT, MPI_FLOAT, MPI_DOUBLE



- Tags
  - Point-to-point kann mit einem benutzerdefinierten Parameter getaggt werden
    - Was für eine Nachricht ist das? Wie soll ich reagieren?

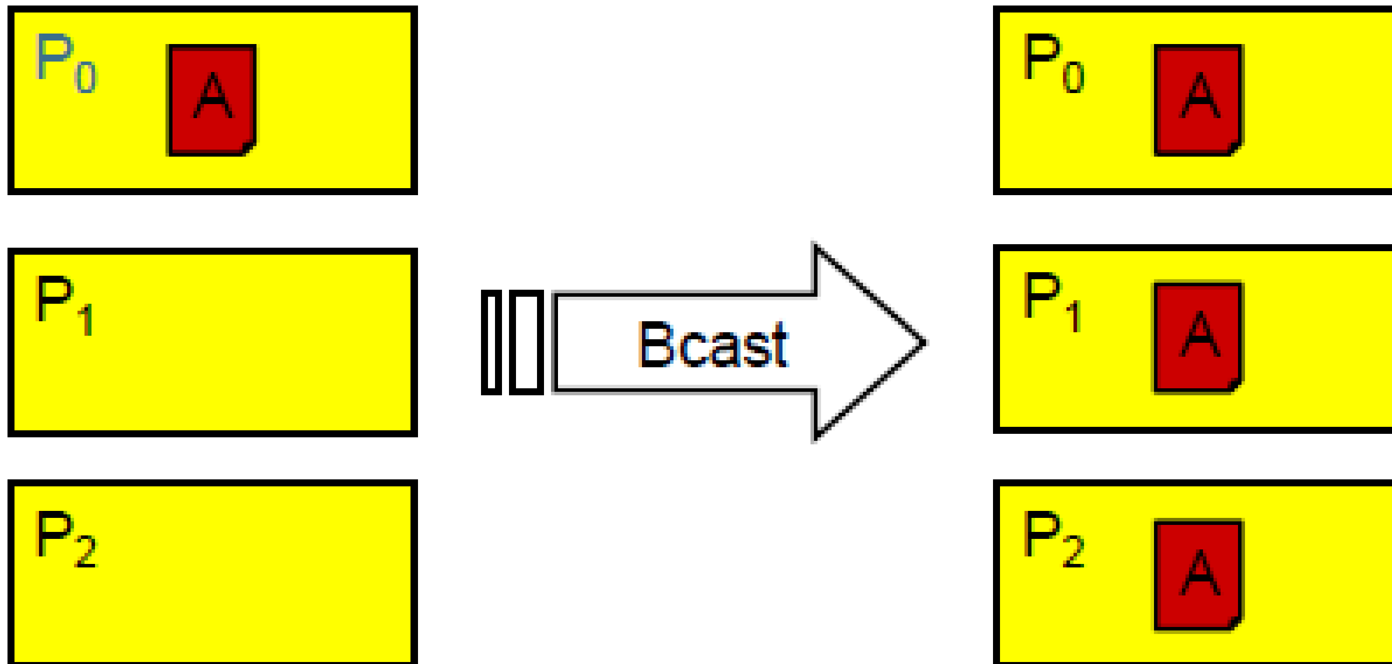


- Nachrichten sind relativ zum aktuellen Communicator

- MPI_Send und MPI_Receive

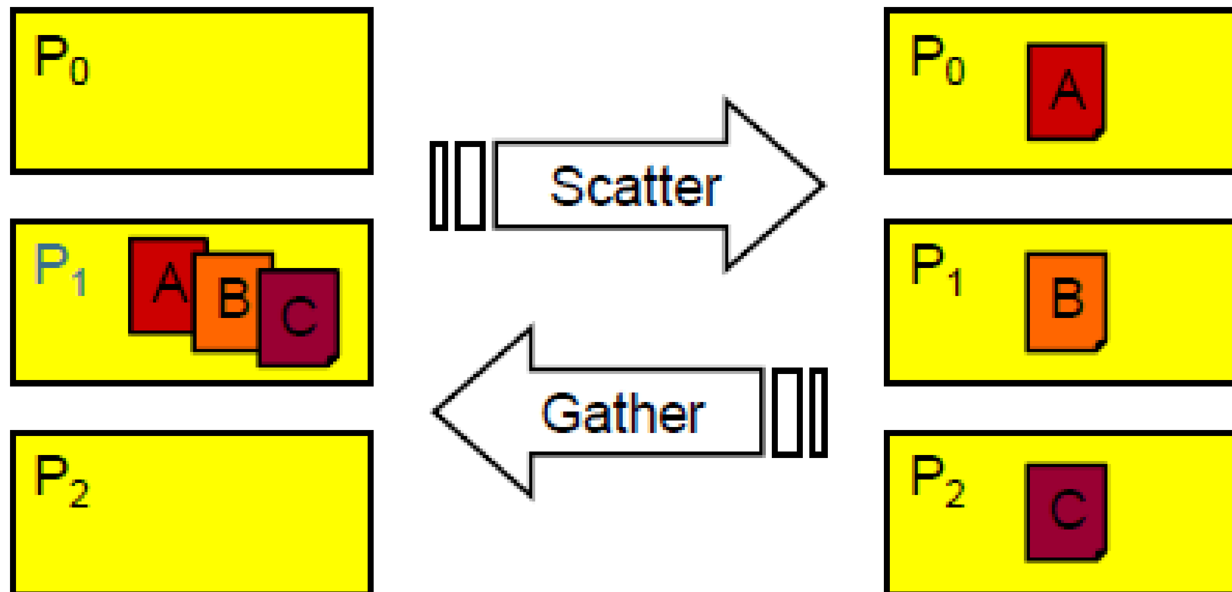
MPI Collectives

- MPI_Barrier
  - Synchronisiert alle Prozesse in einem Communicator
- MPI_Bcast
  - Verteilt Daten von einem Prozessor an alle anderen

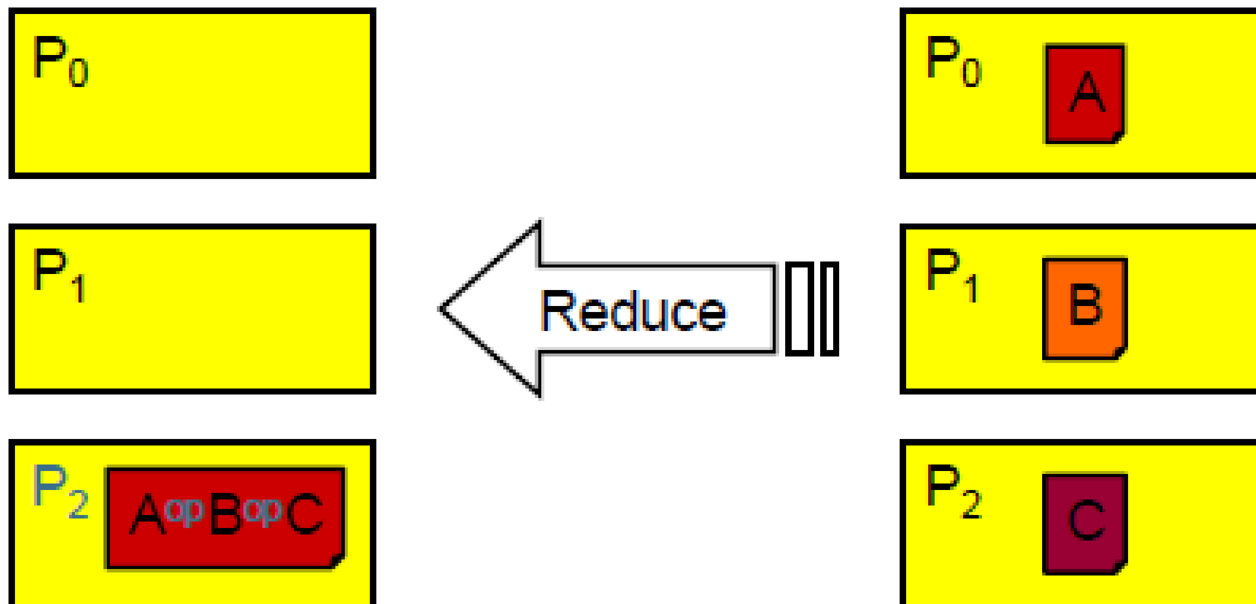


MPI Collectives

- MPI_Scatter
  - Bcast für verschiedene Daten
- MPI_Gather
  - Daten einsammeln



- MPI_Reduce und MPI_Allreduce
  - Daten mit assoziativer Funktion einsammeln
  - Summe, Max, Min, Norm etc.
  - Allreduce: Alle erhalten das Ergebnis (effizienter als Reduce+Bcast)



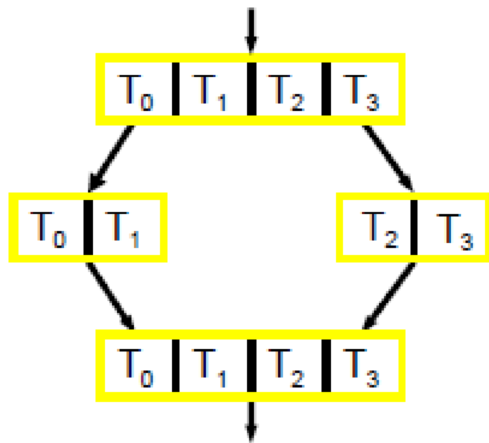
MPI Zusammenfassung

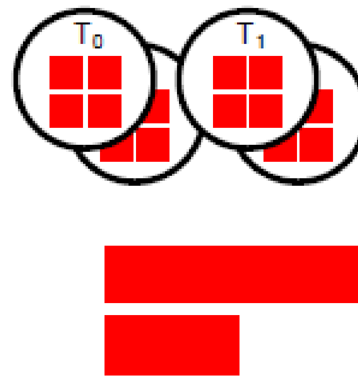
- Vorteile
  - Bietet alles was man braucht
  - Oft keine Alternative
  - Auf Supercomputern optimiert durch die Hersteller, auf meinem Laptop in 10 Minuten installiert (open-source Varianten)
  - 320 Funktionen!
  - Aber man braucht für typische Programme nicht mehr als 10-20
  - Keine steile Lernkurve, wenn man Gebietszerlegung verstanden hat ist die Umsetzung mittels MPI typischerweise nicht mehr schwer
  - Standard, also auf allen distributed memory Architekturen verfügbar
- Nachteile
  - Relativ hoher Programmier-Overhead und viel manuelle Arbeit (explizites Verschicken von Daten)
  - Serieller Referenzcode muss zu Fuss (parallel dazu ☺) auch implementiert werden

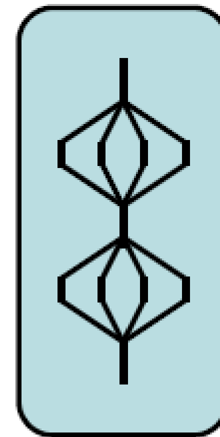
Parallele Programmiermodelle: Threads für shared memory

Shared Memory

- „Sequentieller“ Programmierstil
  - Verteilung der Arbeit auf Threads mittels Fork-Join Prinzip
  - Jeder Thread kann jedes Datum im globalen Speicher lesen und schreiben, jeder Thread hat lokalen, privaten Speicher (scratchpad)
  - KEINE automatische Überprüfung der Direktiven (gleich)
- Standards: POSIX Threads (Windows Threads) und OpenMP







OpenMP Funktionalität

- Direktiven, #pragma's
  - Parallel Regions (gleicher Code parallel ausführen)
  - Parallel Loops (Schleifeniterationen parallel ausführen)
  - Parallel Sections (unterschiedliche Code-Teile parallel ausführen)
  - Diese Dinge werden typischerweise geschachtelt
- Ausführung durch genau einen Master-Thread
- Weitere Funktionen
  - Reduktionen (immer weniger Threads partizipieren)
  - Synchronisation: Barrier, Critical Region, atomare Operationen

Program:

```
a = 1


a = 2


do i = 1,9
   call work(i)
enddo



a = 3
```

(Sequential) Execution:

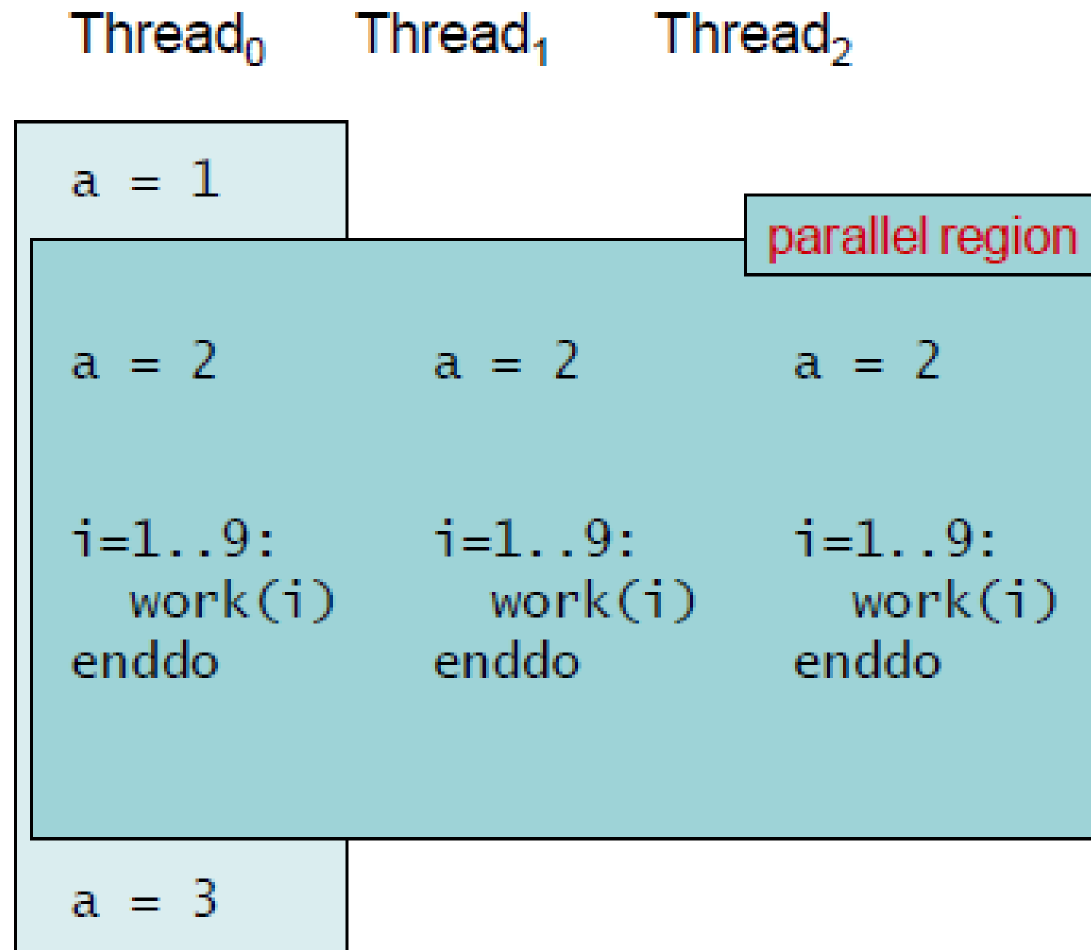
Thread$_0$

```
a = 1


a = 2


i=1..9:
  work(i)
enddo



a = 3
```

Program:

```
a = 1

!$omp parallel
a = 2


do i = 1,9
   call work(i)
enddo

!$omp end parallel

a = 3
```

(Parallel) Execution:

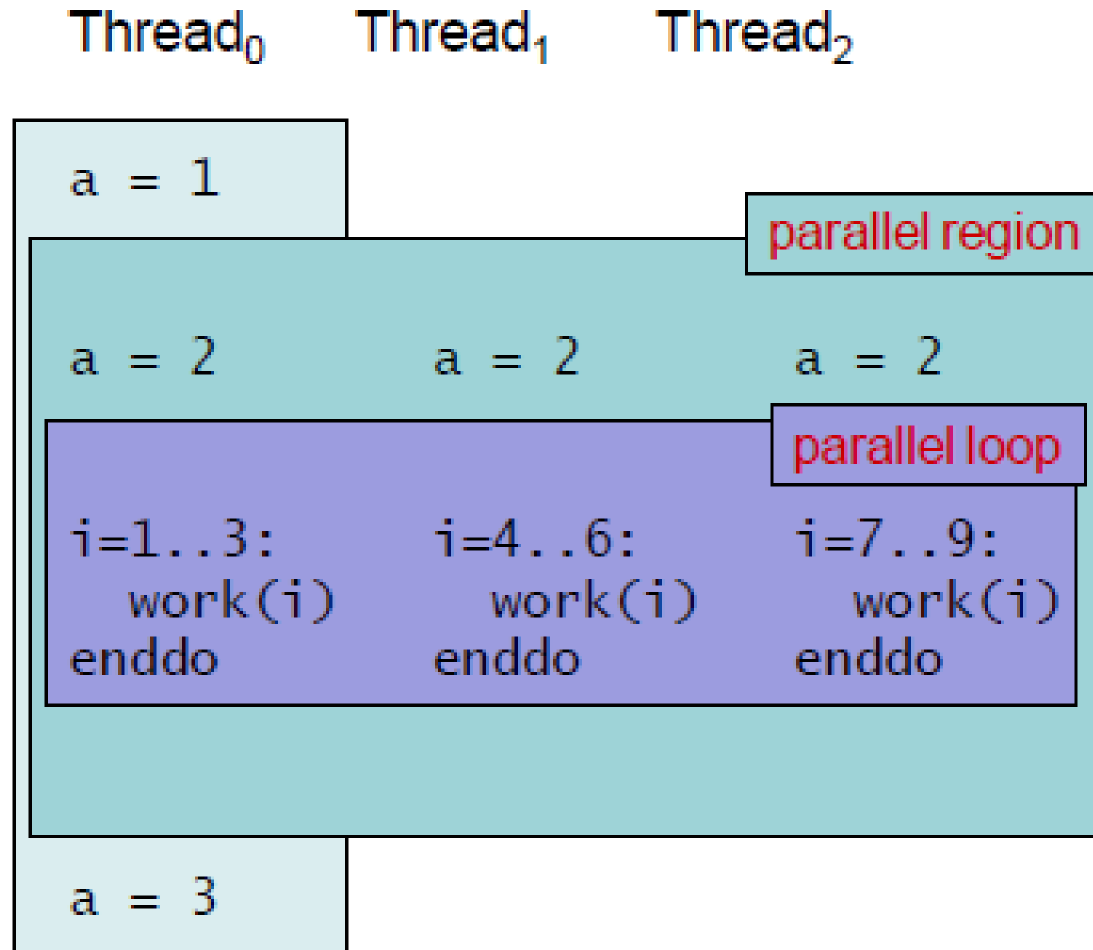


OpenMP Ausführungsmodell

Program:

```
a = 1

!$omp parallel
a = 2

!$omp do
do i = 1,9
   call work(i)
enddo

!$omp end parallel

a = 3
```

(Parallel) Execution:



Hello, World in OpenMP


```
program main
integer :: myid, nthreads
integer :: OMP_Get_num_threads, OMP_Get_thread_num

!$omp parallel private(myid, nthreads)
myid = OMP_Get_thread_num()
nthreads = OMP_Get_num_threads()

write(*,*) "hello from", myid, "of", nthreads
!$omp end parallel

end program
```

Hello, World in OpenMP


```
#include <stdio.h>
#include <omp.h>

int main() {
  int myid, nthreads;

  #pragma omp parallel private(myid, nthreads)
  {
    myid = omp_get_thread_num();
    nthreads = omp_get_num_threads();

    printf("hello from %d of %d\n", myid, nthreads);
  }
}
```

OpenMP Zusammenfassung

- Vorteile
  - Stabiler Standard, fast überall unterstützt
  - Einfache Benutzung über Compilerdirektiven
  - Serieller Code = Paralleler Code
  - „Häppchenweise“ parallelisieren
- Nachteile
  - Beschränkt auf shared memory Architekturen (logisch)
  - Synchronisation bei komplexen Kommunikationsmustern manchmal schwierig
  - Weil einige Code-Teile sequentiell bleiben, ist Speedup beschränkt (Amdahl)