OpenMP Data Scope Clauses

- *private* (list)

  erklärt die Variablen in der Liste list zu privaten Variablen der einzelnen Threads des Teams

- *shared* (list)

  erklärt die Variablen in der Liste list zu gemeinsamen Variablen aller Threads des Teams

- ohne explizite Vereinbarung: Default shared aber
  - Stack- (lokale) Variablen in einem aufgerufenen Unterprogramm sind *private*
  - die Schleifenvariable der parallelen *for*-Schleife ist immer *private*

Example: Calculation of Pi

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4  #include <omp.h>
5
6  #define PI 3.14159265358979323L
7
8  double f (double a)
9  {
10  return (double)4.0/((double)1.0+(a*a));
11 }
12
13 int main(int argc, char *argv[])
14 {
15  long  n, i;
16  double   h, pi, sum, x;
17 for (;;) {
18   printf("Enter the number of intervals: (0 quits) ");
19   scanf("%u",&n);
20   if (n==0) break;
21   h = ((double)1.0)/(double)n;
22   sum = 0.0;
23
24 #pragma omp parallel for private(i,x) reduction(+:sum)
25     for(i=1;i<=n;i++) {                    /* parallel loop */
26       x = h*((double)i-(double)0.5);
27       sum += f(x);
28     }
29                                            /* end omp parallel for */
30
31   pi = h*sum;
32   printf("pi is approximatly: %.16f  Error is: %.16f\n",
33          pi, fabs(pi-PI));
34 }
35
36  return EXIT_SUCCESS;
37 }
```

Weitere Direktiven

- *critical* Direktive
  - Beschränkung des Zugriffs auf den eingeschlossenen Code auf nur einen Thread zur selben Zeit

    *#pragma omp **critical** [(name)]*
    structured block
  - Jeder Thread wartet am Anfang einer *critical*-Region bis kein anderer Thread des Teams die Region dieses Namens ausführt. Alle unbenannten Regionen *critical*-Anweisungen zeigen auf den selben unspezifizierten Namen

- *parallel for* Direktive
  - Parallele Region mit nur einer einzigen parallelen *for*-Schleife

  *#pragma omp **parallel for** [clause [[,] clause] ...]*
  for loop

Paderborn Center for Parallel Computing

Weitere Direktiven

- *single* Direktive
  - Ausführung eines Teils einer parallelen Region durch nur einen zwar den schnellsten Thread:

    *#pragma omp **single***
    structured block

- *barrier* Direktive
  - Synchronisation innerhalb einer parallelen Region

  *#pragma omp **barrier***

- *sections* Direktive
  - Synchronisation innerhalb einer parallelen Region
  - Verteilt unabhängige Programmteile auf Threads.

  *#pragma omp **sections** [clause [[,] clause] ...]*
  {
  structured block 1
  structured block 2
  }

Paderborn Center for Parallel Computing

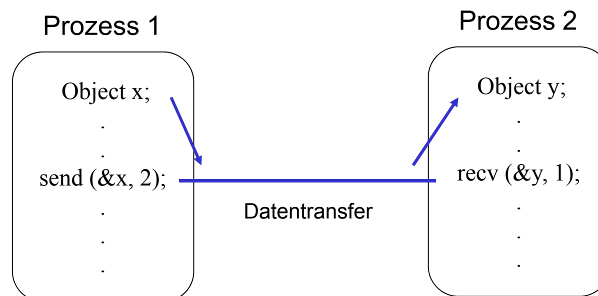
Parallele Bibliotheken: Beispiele

- Intel-Bibliotheken:
  - Math Kernel Library
    - mathematische Funktionen der Linearen Algebra
    - über OpenMP parallelisierte Routinen
    - Funktionen: Matrix-Operationen, Löser für Gleichungssysteme, FFTs, ...
  - Threading Building Blocks
    - Konstrukte: parallel_for, parallel_do, parallel_reduce, pipeline, parallel_sort, parallel_scan
- NAG-Bibliothek:
  - Differentialgleichungen, Optimierung, Lineare Algebra, statistische Methoden, ...

Message-Passing: Einführung

„Punkt zu Punkt“ Senden und Empfangen

- Message-Passing Mechanismus
  - Erzeugung von Prozessen.
  - Übertragung einer Nachricht zwischen den Prozessen durch *send()* und *recv()* Funktionen.



Message Passing Interface (MPI)

- Bibliothek (C, C++, Fortran) für den Austausch von Nachrichten zwischen Prozessen.
- Prozesse werden statisch allokiert und mit 0,…,N-1 nummeriert.
- Jeder Prozess startet das gleiche Programm (Single Program Multiple Data - SPMD).
- Stellt Funktionen und Datentypen beginnend mit „*MPI_*“ bereit.
- (logische) Topologie ist eine Clique
  - Jeder Prozessor kann direkt mit jedem anderen kommunizieren
- Kommunikatoren regeln die Kommunikation zwischen den Prozessen
  - Jeder Prozess gehört zu mindestens einem Kommunikator
  - Jede Nachricht wird in einem Kommunikator transportiert
  - Kommunikatoren können beliebig neu erzeugt werden

MPI Versionen

- Version 1.0: (1994)
  - 129 MPI-Funktionen, C und Fortran77
- Version 1.1
  - Spezifikation verfeinert
- Version 1.2
  - Weitere Eindeutigkeiten
- Version 2.0: (1997)
  - Mehr Funktionalitäten, 193 weitere Funktionen
    - Dynamische Generierung von Prozessen, Einseitige Kommunikation, MPI-Funktionen für I/O
  - C++ und Fortran90
- Version 3.0: (ab 2013)
  - nicht blockierende kollektive Funktionen, Remote Memory Access, nicht blockierende parallele File I/O

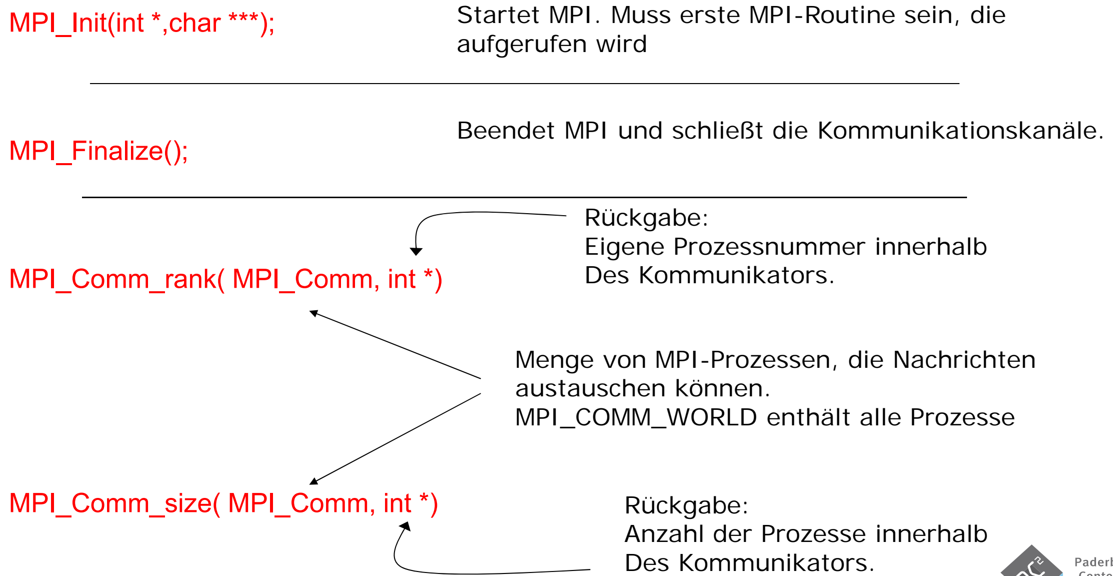
SPMD Ausführungsmodell

Single Program Multiple Data (SPMD) lässt sich aber auch erweitern:

```
#include <mpi.h>
main (int argc, char *argv[])
{
    MPI_Init(&argc, &argv);
    …
    MPI_Comm_rank(MPI_COMM_WORLD, & myrank);  /* find process rank */
    if (myrank == 0)
        master();
    else
        worker();
    ...
    MPI_Finalize();
}
```

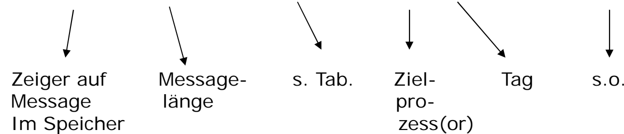
- Dabei sind master() und worker() Funktionen die durch einen Master- und ggf. mehrere Worker-Prozesse ausgeführt werden sollen.

Basisroutinen



Basisroutinen

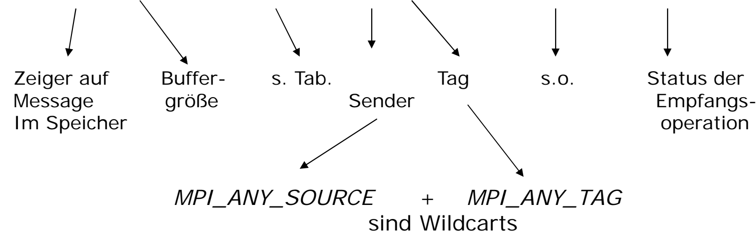
MPI_Send(void *, int, MPI_Datatype, int, int, MPI_Comm);



MPI_Datatype	C-Datentyp
MPI_CHAR	char
MPI_SHORT	short
MPI_INT	int
MPI_LONG	long
MPI_UNSIGNED_CHAR	(unsigned) char
MPI_UNSIGNED_SHORT	unsogned short
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_DOUBLE_LONG	long double
MPI_BYTE	-
MPI_PACKED	-

Basisroutinen

MPI_Recv(void *, int, MPI_Datatype, int, int, MPI_Comm, MPI_Status*);



- Tag und Communicator des MPI_Send und MPI_Recv müssen übereinstimmen
- MPI_Status enthält u.a.:
  status→MPI_SOURCE (Sender der Nachricht)
  status→MPI_TAG (Message-Tag)
  status→MPI_ERROR (0, falls kein Fehler)
- MPI_Send/MPI_Recv blockiert (eventuell!)

Paderborn Center for Parallel Computing

MPI für Java

- Objekt-orientiertes Java-Interface für MPI entwickelt im HPJava Projekt
- mpiJava 1.2 unterstützt MPI 1.1 (z.B. MPICH)
  - http://www.hpjava.org/mpiJava.html

```
import mpi.*;
public class MPI_HelloWorld
{
   public static void main(String[] args)
   {
      System.out.println(„Hello, world“);
   }
}
```

Ausführung
- javac MPI_HelloWorld.java
- prunjava 4 MPI_HelloWorld

openMPI unterstützt ebenfalls Java

Paderborn Center for Parallel Computing

MPI Java Example

```
public static void main(String args[]) throws MPIException
{
   MPI.Init(args);

   int rank = MPI.COMM_WORLD.getRank(),
      size = MPI.COMM_WORLD.getSize(),
      nint = 100; // Intervals.
   double h = 1.0 / (double)nint,
        sum = 0.0;
   for (int i = rank + 1; i <= nint; i += size) {
      double x = h * ((double)i - 0.5);
      sum += (4.0 / (1.0 + x * x));
   }

   double sBuf[] = { h * sum }, rBuf[] = new double[1];

   MPI.COMM_WORLD.reduce(sBuf, rBuf, 1, MPI.DOUBLE, MPI.SUM, 0);
   if (rank == 0) System.out.println("PI: "+ rBuf[0]);

   MPI.Finalize();
}
```


Further MPI bindings

- MPI for Python
  - http://pympi.sourceforge.net/
  - http://mpi4py.scipy.org/
- and more
  - Common LISP
  - Ruby
  - Perl
  - Caml
  - Common Language Infrastructure (CLI)

Kommunikationsmethoden

buffered / non-buffered

- Nachricht wird (beim Sender) zwischengespeichert, oder
- keine Zwischenspeicherung der Nachricht

synchron / nicht synchron

- Programmflusskontrolle wird solange nicht zurückgegeben, bis die zu sendende Nachricht angekommen ist, oder
- kein Warten auf den Empfänger

blockierend / nicht blockierend

- Programmflusskontrolle wird nicht zurückgegeben, bis die zu sendende Nachricht entweder beim Empfänger oder im eigenen Buffer gesichert ist, oder
- Funktionsaufruf kommt unmittelbar zurück

Achtung: Nach Rückkehr des non-blocking Send kann die zu sendende Nachricht ggf. verändert werden

MPI Send Kommunikationsmodi (1)

- Standard Mode Send
  - passendes Recv muss nicht vor Send ausgeführt werden.
  - Größe des Buffers ist nicht im MPI-Standard definiert.
  - Falls Buffer genutzt, kann das Send beendet werden bevor das passende Recv erreicht ist.
  - Falls kein Buffer genutzt, dann warten auf entsprechendes Recv.
- Buffered Mode
  - Send kann vor dem passenden Recv starten und beenden.
  - Speicher für Buffer muss durch Sender explizit über *MPI_Buffer_attach()* allokiert werden .
- Synchronous Mode
  - Send und Recv können beliebig starten, beenden aber gemeinsam.
- Ready Mode
  - Send kann nur starten, wenn passendes Recv bereits erreicht wurde. Ansonsten Rückgabe einer Fehlermeldung.

MPI Send Kommunikationsmodi (2)

- Alle vier Modi können mit einem blocking oder non-blocking Send kombiniert werden.

Semantik	Standard	Synchrounous	Buffered	Ready
Blocking	MPI_Send	MPI_Ssend	MPIBsend	MPI_Rsend
Non-blocking	MPI_Isend	MPI_Issend	MPI_Ibsend	MPI_Irsend

- Jeder Send Typ kann mit einem blocking oder non-blocking Recv matchen.

- Kombiniertes Senden und Empfangen
  MPI_Sendrecv()

MPI Send Modes

- **MPI_Send** will not return until you can use the send buffer. It may or may not block (it is allowed to buffer, either on the sender or receiver side, or to wait for the matching receive).
- **MPI_Bsend** May buffer; returns immediately and you can use the send buffer. A late add-on to the MPI specification. Should be used only when absolutely necessary.
- **MPI_Ssend** will not return until matching receive posted
- **MPI_Rsend** May be used ONLY if matching receive already posted. User responsible for writing a correct program.
- **MPI_Isend** Non-blocking send. But not necessarily asynchronous. You can NOT reuse the send buffer until either a successful, wait/test or you KNOW that the message has been received (see MPI_Request_free).
- **MPI_Ibsend** buffered and non-blocking
- **MPI_Issend** Synchronous non-blocking. Note that a Wait/Test will complete only when the matching receive is posted.
- **MPI_Irsend** As with MPI_Rsend, but non-blocking.

Beispiel: Blockierendes Senden

Blockierendes Senden eines Werts x von MPI-Prozess 0 zu MPI-Prozess 1

```
int myrank;
int msgtag = 4711;
int x;

…
    MPI_Comm_rank (MPI_COMM_WORLD, & myrank);  /* get process rank */
    if (myrank == 0)
       MPI_Send (&x, 1, MPI_INT, 1 , msgtag, MPI_COMM_WORLD);
    else if (myrank == 1) {
       int status;
       int x;
       MPI_Recv (&x, 1, MPI_INT, 0, msgtag, MPI_COMM_WORLD, status);
    }
...
```


Non-Blocking Routinen

- Non-blocking Send:
  - *MPI_Isend(buf, count, datatype, dest, tag, comm, request)*
  - Kommt sofort zurück, obwohl die zu versendenden Daten noch nicht geändert werden dürfen.
  - **Nicht unbedingt ein asynchrones send**
- Non-blocking Receive:
  - *MPI_Irecv(buf, count, datatype, dest, tag, comm, request)*
  - Kommt sofort zurück, obwohl ggf. noch keine Daten vorliegen

Terminierungserkennung durch *MPI_Wait()* und *MPI_Test()*

- *MPI_Wait()* wartet bis Operation beendet ist
- *MPI_Test()* kommt sofort mit dem Zustand der Send- / Recv-routine (beendet bzw. nicht-beendet) zurück
- Dafür muss der request Parameter verwendet werden

Beispiel: Nichtblockierendes Senden

Nichtblockierendes Senden eines Werts x von MPI-Proz. 0 zu MPI-Proz. 1, wobei MPI-Prozess 0 mit Ausführung unmittelbar fortfahren kann.

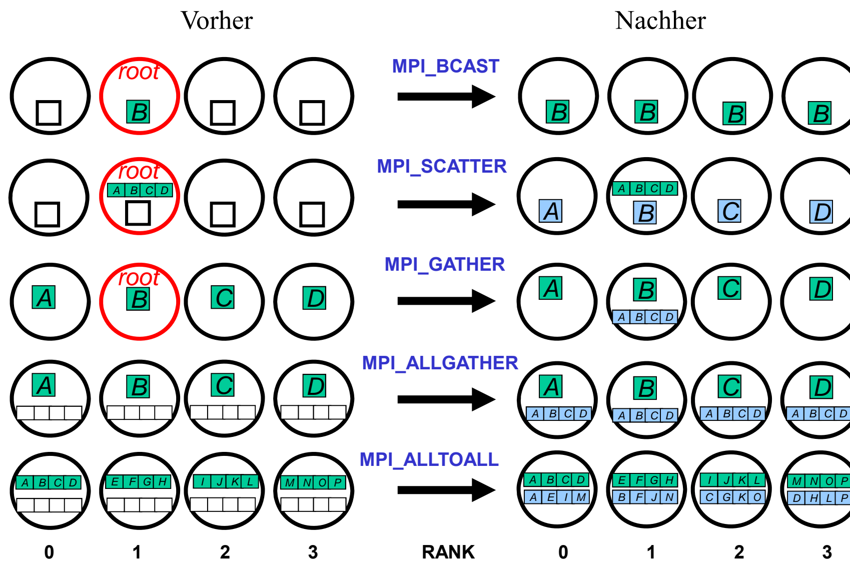
```
int myrank;
int msgtag = 4711;
…
   MPI_Comm_rank (MPI_COMM_WORLD, & myrank);                    /* find process rank */
   if (myrank == 0) {
      int status;
      int x;
      MPI_Isend (&x, 1, MPI_INT, 1 , msgtag, MPI_COMM_WORLD, req1);
      compute();
      MPI_Wait (req1, status);
   } else if (myrank == 1) {
      int status;
      int x;
      MPI_Recv (&x, 1, MPI_INT, 0, msgtag, MPI_COMM_WORLD, status);
   }
    ...
```


Kollektive Kommunikation

- Kommunikation innerhalb einer Gruppe aus Prozessen
- keine Message-Tags nutzbar

- Broadcast- und Scatter-Routinen
  - *MPI_Bcast()* - Broadcast from root to all other processes
  - *MPI_Gather()* - Gather values for group of processes
  - *MPI_Scatter()* - Scatters buffer in parts to group of processes
  - *MPI_Alltoall()* - Sends data from all processes to all processes
  - *MPI_Reduce()* - Combine values on all processes to single value
  - *MPI_Reduce_Scatter()* - Combine values and scatter results
  - *MPI_Scan()* - Compute prefix reductions of data on processes

Kollektive Kommunikation



Beispiel: MPI_Gather

Beachte, dass *MPI_Gather* von **allen** Prozessen **inkl.** Root aufgerufen werden muss!

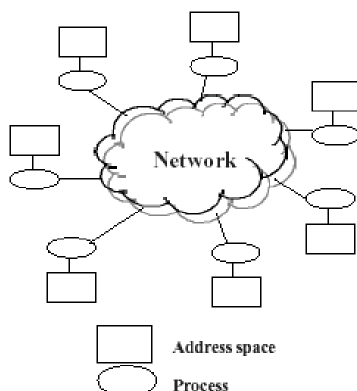
```
int data[10];                                  /* data to be gathered from processes */

…
   MPI_Comm_rank (MPI_COMM_WORLD, & myrank);          /* find process rank */
   if (myrank == 0) {
      MPI_Comm_Size(MPI_COMM_WORLD, &grp_size);
      buf = (int*)malloc(grp_size*10*sizeof(int));    /* allocate memory /*
   }
   MPI_Gather (data, 10, MPI_INT, buf, grp_size*10, MPI_INT, 0, MPI_COM_WORLD);
…
```

Einseitige Kommunikation

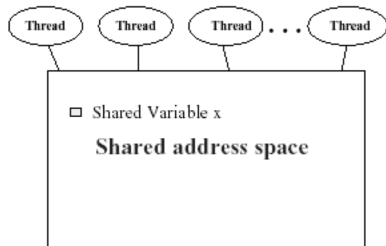
- Verfügbar ab MPI Version 2
- Remote Memory Access, put + get Operationen
- Initialisierungen
  - *MPI_Alloc_mem(), MPI_Free_mem()*
  - *MPI_Win_create(), MPI_Win_free()*
- Kommunikationsroutine
  - *MPI_Put()*
  - *MPI_Get()*
  - *MPI_Accumulate()*
- Synchronizationen
  - *MPI_Win_fence()*
  - *MPI_Win_post(), MPI_Win_start(), MPI_Win_complete(), MPI_Win_wait()*
  - *MPI_Win_lock(), MPI_Win_unlock()*

Message-Passing-Modell



- Pro's:
  - Programmierer kontrolliert die Daten- und Arbeitsverteilung
  - Eindeutigkeit der Lokalität
- Con's:
  - Relativ hoher Kommunikationsaufwand bei kleinen Nachrichten
  - Korrektheit der Programme schwer zu überprüfen
- Beispiel:
  - Message Passing Interface (MPI)

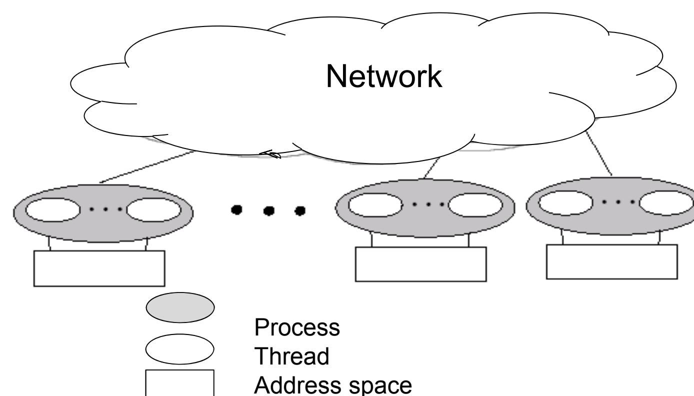
Shared-Memory-Modell



- Pro's:
  - Einfache Befehle
    Lesen / Schreiben des entfernten Speichers i.W. durch einfache Wertzuweisung
  - Programmierwerkzeuge vorhanden
- Con's:
  - Manipulation von gemeinsam genutzten Speicherzellen erfordern meistens doch explizite Synchronisation
  - Keine Eindeutigkeit der Lokalität
- Beispiele:
  - OpenMP, POSIX Threads

Paderborn Center for Parallel Computing

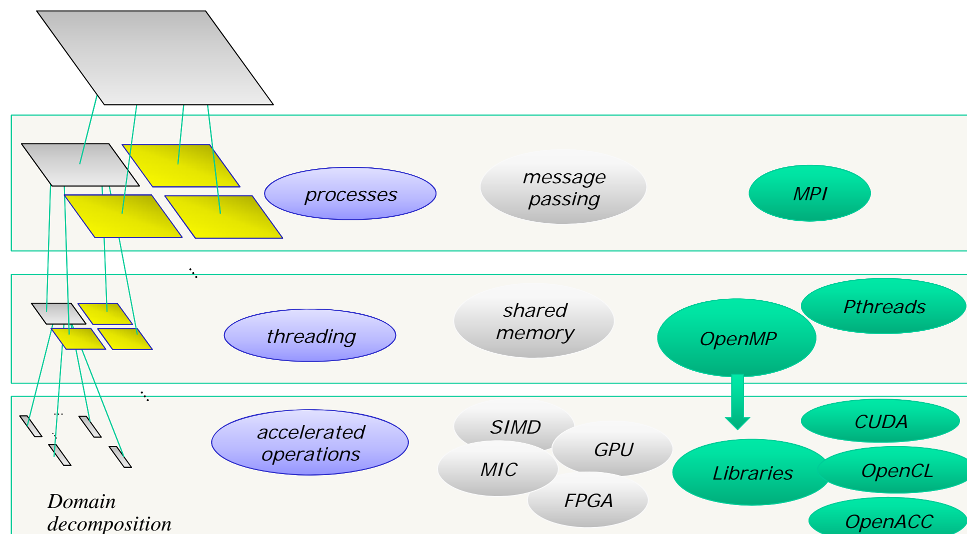
Hybrides Modell: Shared Mem. + Message Passing



- Beispiele:
  - POSIX Threads innerhalb der Knoten und MPI zwischen Knoten
  - OpenMP für Intra- und MPI für Inter-Knoten-Kommunikation

Paderborn Center for Parallel Computing

Hybride Programmierung

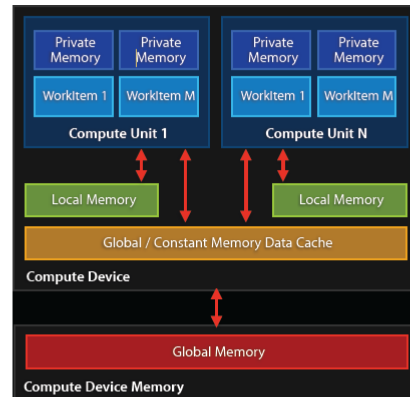


OpenCL

- Open Computing Language
  - parallel execution on single or multiple processors
  - for heterogeneous platforms of CPUs, GPUs, and other processors
  - Desktop and handheld profiles
  - works with graphics APIs such as OpenGL
  - based on a proposal by Apple Inc.
  - supported processors: Intel, AMD, Nvidia, and ARM
  - work in progress: FPGA
- Design Goals
  - Data- and task-parallel compute model based on C99 language
  - Implements a relaxed consistency shared memory model with multiple distinct address spaces
- OpenCL 2.0
  - Device partitioning, separate compilation and linking, Enhanced image support, Built-in kernels, DirectX functionality

OpenCL

- Implements a relaxed consistency shared memory model
  - Multiple distinct address spaces
  - Address space can be collapsed depending on the device's memory subsystem
  - Address qualifiers
    - *__private*
    - *__local*
    - *__constant* / *__global*
    - Example: *__global float4 *p;*
- Built-in Data Types
  - Scalar and vector data types
  - Pointers
  - Data-type conversion functions
  - Image type (2D/3D)



OpenACC

- Open Accelerator
  - "High level gateway to lower level CUDA GPU programming language"
  - accelerate C and FORTRAN code
  - directives identifies parallel regions
  - Initially developed by PGI, Cray, Nvidia
  - supported processors: AMD, NVIDIA, Intel?
- Design Goals
  - No modification or adaption of programs to use accelerators
- OpenACC compiler
  - First compilers from Cray, PGI, and CAPS
  - GCC ab Version 5 (offloading to Nvidia targets)

MPI + X

X = {OpenMP, Pthreads, spezielle Bibliotheken}

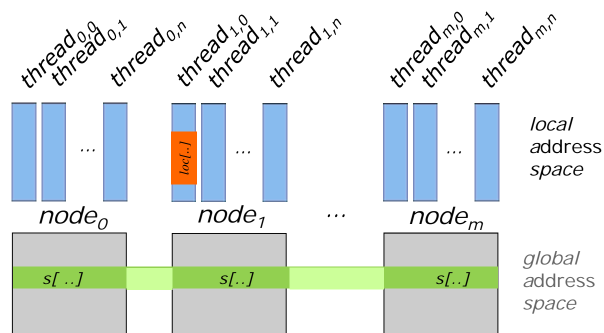
- Pros
  - Reduzierung der Anzahl an MPI-Prozesse
    - dadurch ggf. bessere Skalierung
  - Weniger Overhead für die Zwischenspeicherungen von Daten
  - Schnellere Synchronisation innerhalb (Unter-) Gruppen
- Cons
  - Zwei unterschiedliche Programmierparadigmen
  - Portabilität könnte problematisch sein

Paderborn Center for Parallel Computing

Partitioned Global Address SPACE

- PGAS parallel programming model
- Global memory address space that is logically partitioned
  - A portion of the memory is local to each process or thread
  - Combines advantages of SPMD programming for distributed memory systems with data referencing semantics of shared memory systems

- *Variables and arrays can be shared or local*
- *Thread can use references to its local variables and all shared variables*
- *Thread has fast access to its local variables and its portion of shared variables*
- *Higher latencies to remote portions of shared variables*



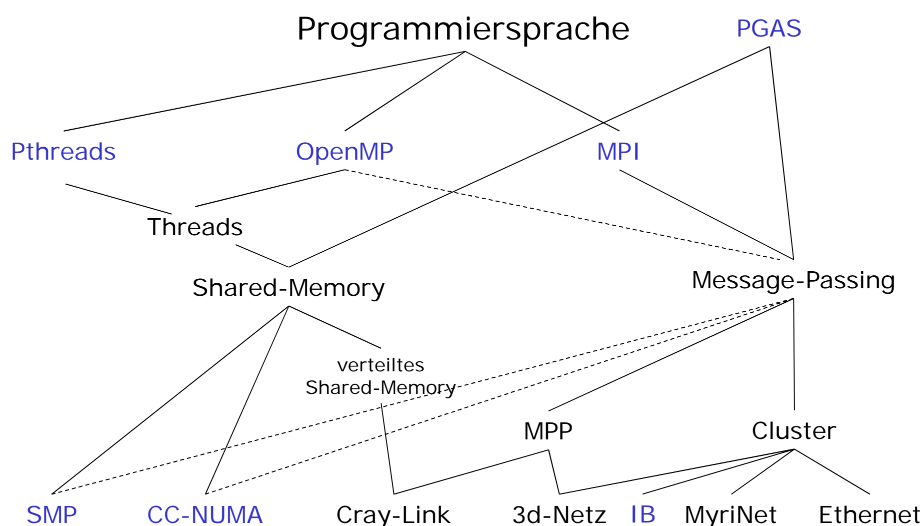
Paderborn Center for Parallel Computing

Implementations of PGAS

- PGAS fits well to the communication hierarchy of hybrid architectures
  - Communication network, shared memory nodes
- Affinity of threads to parts of the global memory effects the efficiency of program execution

- Languages
  - Unified Parallel C http://upc.lbl.gov
  - Co-array Fortran http://www.co-array.org/
  - Titanium http://titanium.cs.berkeley.edu/
  - Fortress http://labs.oracle.com/projects/plrg/
  - Chapel http://chapel.cray.com/
  - X10 http://x10-lang.org/

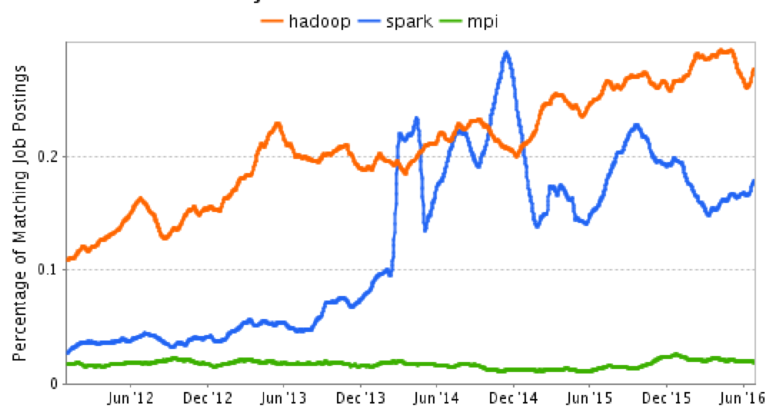
Abstraktionsebenen der Programmierung



Parallelarbeit vs. Parallelitätsebenen

Parallelarbeitstechniken	Suboperationsebene	Anweisungsebene	Blockebene	Prozeßebene	Programmebene
SIMD-Techniken					
Vektorrechnerprinzip	X				
Feldrechnerprinzip	X				
Techniken der Parallelarbeit in der Prozessorarchitektur					
Befehlspipelining		X			
Superskalar		X			
VLIW		X			
Überlappung von E/A- mit CPU-Operationen		X			
Feinkörniges Datenflußprinzip		X			
Techniken der Parallelarbeit durch Prozessorkopplung					
Speicherkopplung (SMP)			X	X	X
Speicherkopplung (DSM)			X	X	X
Grobkörniges Datenflußprinzip			X	X	
Nachrichtenkopplung				X	X
Techniken der Parallelarbeit durch Rechnerkopplung					
Workstation-Cluster				X	X
Grid- und Cloud-Computer				X	X

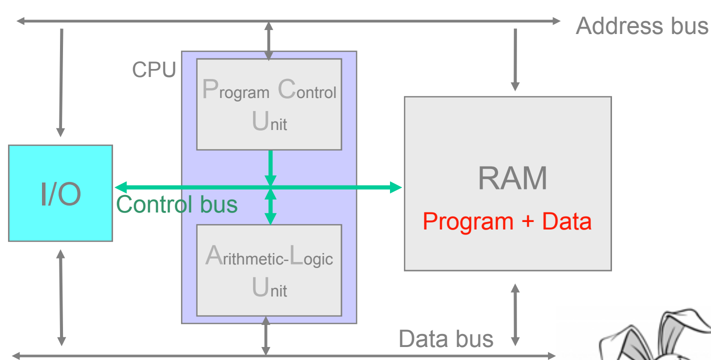
What are users doing?



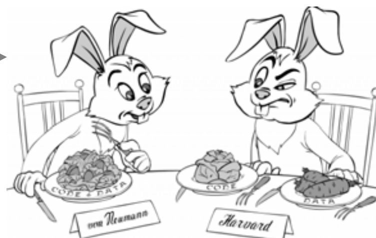
Source: Indeed.com

Grundlagen der Rechnerarchitektur

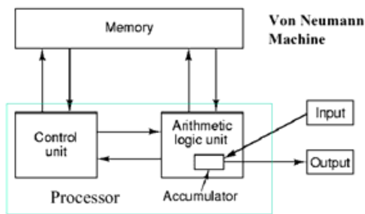
Von Neumann Architecture

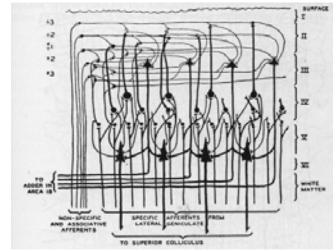


1947: John von Neumann developed a program controlled universal computer engine



The von Neumann Computer

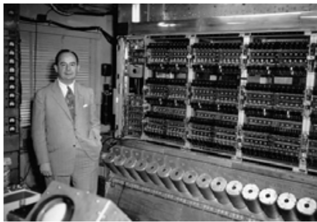




Simple Operation

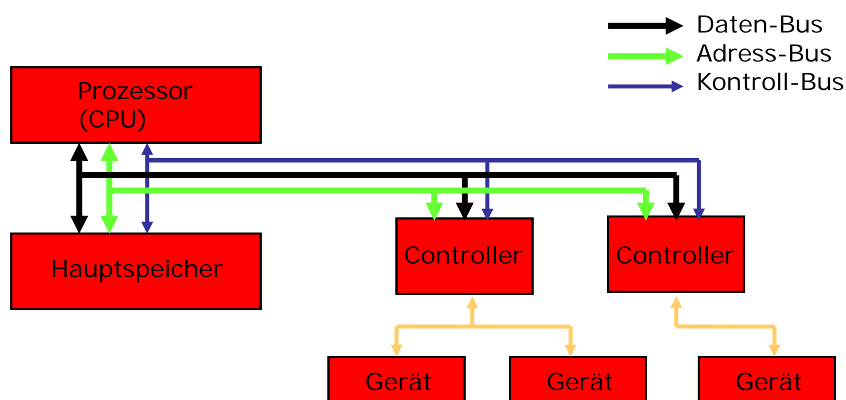
c = a + b

1. *Get first instruction*
2. *Decode: Fetch **a***
3. *Fetch **a** to internal register*
4. *Get next instruction*
5. *Decode: fetch **b***
6. *Fetch **b** to internal register*
7. *Get next instruction*
8. *Decode: add **a** and **b** (**c** in register)*
9. *Do the addition in ALU*
10. *Get next instruction*
11. *Decode: store **c** in main memory*
12. *Move **c** from internal register to main memory*



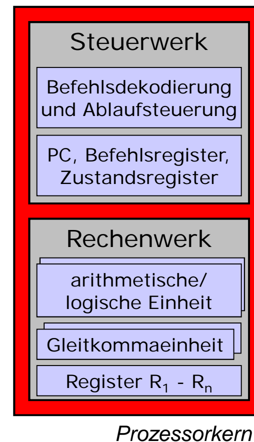
Execution is strong sequential

Nach wie vor eingesetzte Architektur

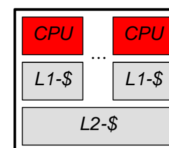


Prozessor

- Grundelemente eines Prozessors sind
  - **Rechenwerk**: führt die arithmetischen und die logischen Operationen aus
  - **Steuerwerk**: stellt Daten für das Rechenwerk zur Verfügung, d.h. holt die Befehle aus dem Speicher, koordiniert den internen Ablauf
  - **Register**: Speicher mit Informationen über die aktuelle Programmbearbeitung, z.B.
    - Rechenregister, Indexregister
    - Stapelzeiger (stack pointer)
    - Basisregister (base pointer)
    - Befehlszähler (program counter, PC)
    - Unterbrechungsregister,...
- Moderne Prozessoren bestehen aus Prozessorkernen und Caches
- Jeder Prozessorkern hat mehrere Rechenwerke (Funktionseinheiten)



Prozessorkern



Prozessor

Paderborn Center for Parallel Computing

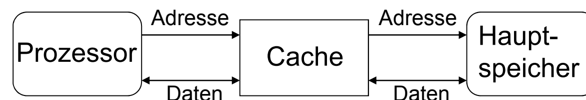
Hauptspeicher

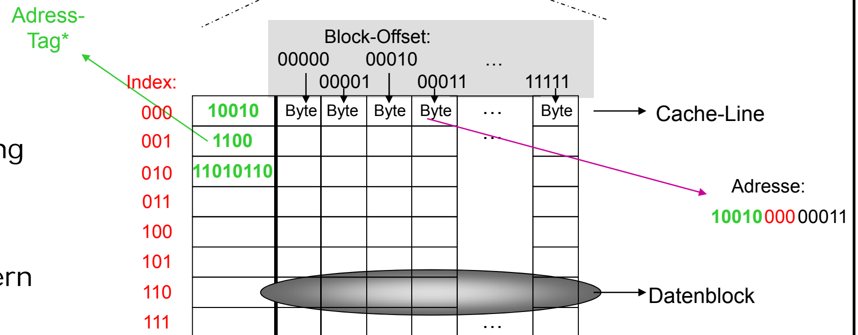
- Hauptspeicher (Arbeitsspeicher):
  - flüchtige Speicherung der aktiven Programme und der dazugehörigen Daten
- typische Größe derzeit bei PCs
  - 4 GByte bis 64 GByte
- Organisation des Speichers
  - Folge aus Bytes, auf die einzeln lesend oder schreibend zugegriffen werden kann (Random Access Memory, RAM)
  - theoretische Größe des Speichers wird durch die Breite des Adressbusses festgelegt
- Virtualisierung des Speichers
  - einheitliche logische Adressräume
  - effizienter Nutzung des physischen Speichers

0	
1	
2	
3	
. . .	
max	

Paderborn Center for Parallel Computing

Daten-Cache



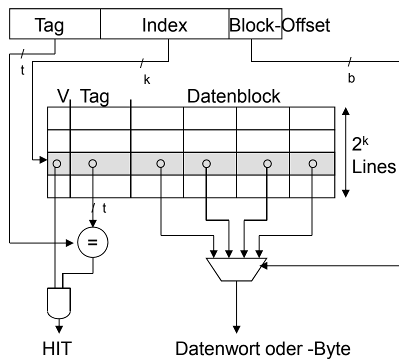


Die Geschwindigkeit des Hauptspeichers ist zu gering für die Rechenleistung der Prozessoren

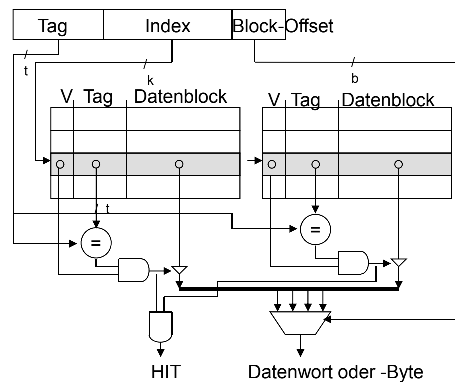
⇒ Einsatz von Cache-Speichern notwendig

Assoziativität eines Caches

Direct-mapped Cache



2-Way Set-Associative Cache



Cache-Aufbau und Charakteristika

- Cache-Lines (Blockgröße z.B. 64 Bytes)
- Caches deutlich kleiner als Hauptspeicher
  ⇒ mapping von HS-Blöcken notwendig
  - direct mapped ⇒ jeder Block wird auf festen Bereich abgebildet
  - fully associate ⇒ Block kann überall im Cache abgelegt werden
  - m-way set associative:
    - Block kann beliebig in einer Menge von *m* Cache-Lines abgelegt werden
    - Replacement: zufällig oder LRU
    - verallgemeinert die beiden anderen Prinzipien
- Weitere Charakteristika:
  - Latenzzeit
  - Bandbreite
  - Kapazität des Caches

Paderborn Center for Parallel Computing

Cache-Miss-Typen

- Compulsory-Miss:
  - auch *cold-start-miss* genannt
  - erster Zugriff auf Block führt zu einem Cache-Miss
- Capacity-Miss:
  - Cache nicht groß genug für alle benötigten Blöcke
  - Cache-Miss ist aufgetreten, obwohl der Block schon einmal im Cache war und verdrängt wurde
- Conflict-Miss:
  - Adresskollision
  - auch *collision-miss* oder *interferences-miss* genannt
  - „Capacity-Miss" der durch eine zu geringe Assoziativität des Caches oder zu große Blöcke begründet ist
  - „Ping-Pong" Effekt möglich

Paderborn Center for Parallel Computing

Compulsory-Misses

- Problem:
  - Noch nicht genutzte Daten sind typischerweise nicht im Cache
  - Folge: erster Zugriff auf Daten potentiell sehr teuer
- Optimierung:
  - größere Cache-Lines (räumliche Lokalität nutzen)
    - erhöht Latenzzeit bei Cache-Miss
    - bei gleichbleibender Kapazität, kann dies Anzahl der Conflict-Misses erhöhen
  - Prefetching
    - Datenwort laden bevor es tatsächlich benötigt wird
    - Überlappung von Berechnungen und Ladeoperationen
    - nur sinnvoll bei nicht-blockierenden Caches
    - Varianten
      - nächstes Datenwort laden (unit stride access)
      - wiederkehrende Zugriffsmuster erkennen und laden (stream prefetch)
      - Zeigerstrukturen laden

Strategien beim Design

- Latenzzeit bei *Hit* und *Miss* reduzieren
  - Speicherzugriffe überlappend ausführen (Pipelining)
  - Multi-Level-Cache
  - diverse andere Hardwaretechniken (s. Hennessy „Computer Architecture“
- Cache-Trefferrate erhöhen
  - Datenzugriffstransformationen
  - Daten-Layout-Transformationen
  - Spezielle Hardware-Techniken
- Latenzzeit verdecken
  - Out-of-Order Execution
  - Expensives Multithreading

Speicherhierarchien

- *Memory Wall*
  Der Unterschied zwischen Prozessor- und Speichergeschwindigkeit wird immer größer

- Leistungsfähigere Speicherhierarchien notwendig
  – Größere Speicherkapazitäten
    • kleinere Fertigungsstrukturen nutzbar
  – Höhere Frequenzen
    • Steigerung beschränkt durch die „Physik“
  – Breitere Datenpfade
    • beschränkt durch Parallelität in der Anwendung
  – Mehr Speicherebenen
    • Potentiell auf mehreren Ebenen nacheinander folgende Misses