Computergrundlagen
Moderne Rechnerarchitekturen

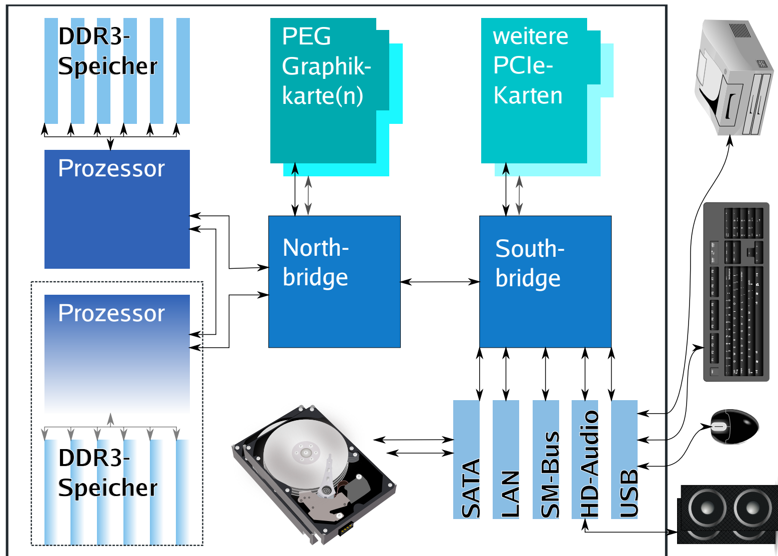
Axel Arnold

Institut für Computerphysik
Universität Stuttgart

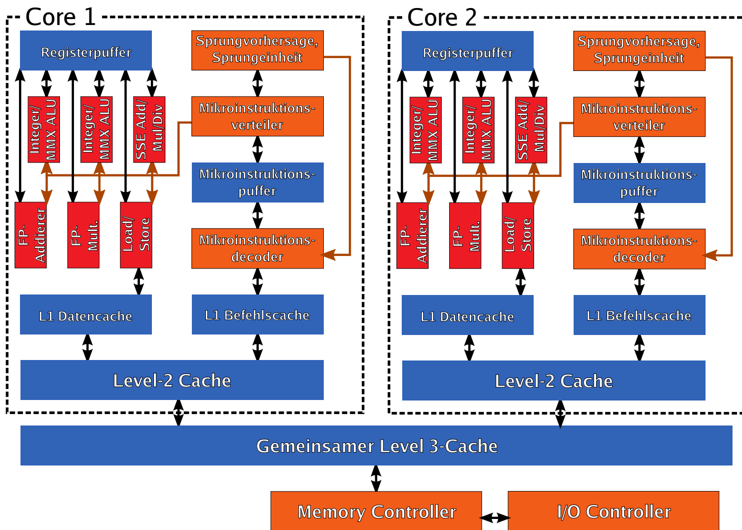
Wintersemester 2010/11

http://www.icp.uni-stuttgart.de

Aufbau eines modernen Computers

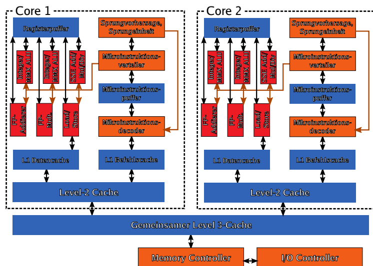


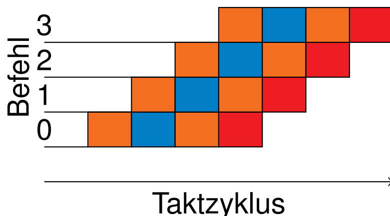
Aufbau eines modernen Prozessors



http://www.icp.uni-stuttgart.de

Befehlsverarbeitung – Pipelines

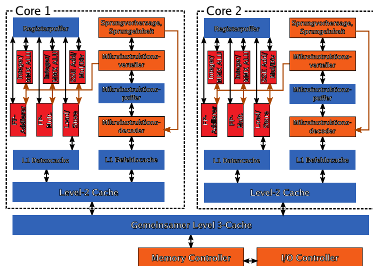




- Während der erste Befehl dekodiert wird, kann der nächste bereits geladen werden
- Analog in den weiteren Stufen bis zu den Funktionseinheiten
- Intel Nehalem: 16-stufige Pipeline, AMD Barcelona: 12-stufig
- Problem: falsche Sprungvorhersage $\rightarrow$ alle angefangenen Befehle verwerfen (Pipeline stall)

http://www.icp.uni-stuttgart.de

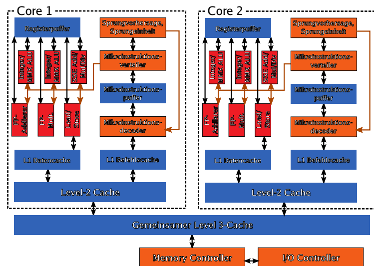
ALUs, FPUs und Co. – wo gearbeitet wird

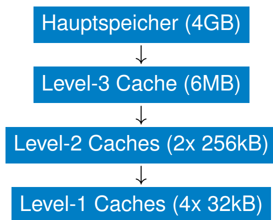


Funktionseinheiten

- Es gibt mehrere Funktionseinheiten (Nehalem: 9, Barcelona: 6)
- Manche Aufgaben können von mehreren übernommen werden, z.B. Ganzzahlrechnungen
- Dadurch können Befehle gleichzeitig bearbeitet werden
- Simultanes Multithreading (Hyperthreading): Zwei Kerne teilen sich dieselben Funktionseinheiten – bessere Auslastung

Speicherpfad – Caches

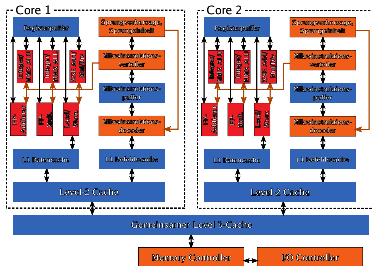


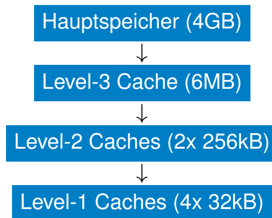


- Taktrate 3GHz →**1 Befehl alle 0.33ns**
- Speicher liefert im günstigsten Fall **alle 10ns neue Daten**
- Caches sind kleiner, aber dafür schneller
- Daten können mehrmals aus dem Cache geladen werden
- Prozessor muss nicht warten, bis Daten zurückgeschrieben
- Daten können im Voraus geladen werden
- Cache lädt ganze Blöcke (Cachelines) von derzeit 64 Byte

http://www.icp.uni-stuttgart.de

Speicherpfad – Caches

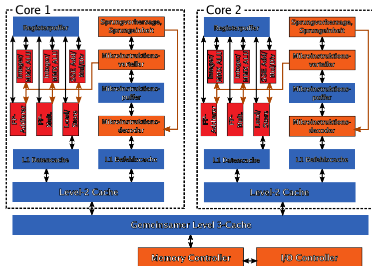




- Caching beim Schreiben von Programmen berücksichtigen
- Einmal geladene Werte möglichst oft wiederverwenden, da diese bereits im Cache liegen
- Möglichst benachbarte Werte lesen und schreiben
- Keine Schleifen, die durch den Speicher „springen“
- Ideal sind Schleifen, die den Speicher linear abarbeiten
- *Beispiel*: Matrizen möglichst zeilenweise bearbeiten

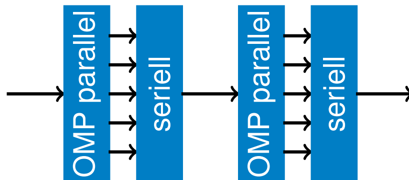
http://www.icp.uni-stuttgart.de

Multicore

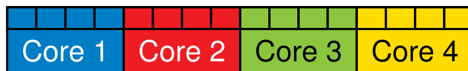


- Alle modernen Prozessoren besitzen mehrere Kerne (Cores)
- Dazu kommen virtuelle Kerne (Simultanes Multithreading)
- Einfache Python- oder C-Programme nutzen nur einen Kern
- Mehrkernsystem erforden neue Algorithmen
- Programmierung z.B. mit OpenMP

OpenMP



- OpenMP: bequeme Programmierung mit mehreren Kernen
- In parallelen Regionen arbeiten alle Kerne des Prozessors – das Programm läuft in mehreren **Threads**
- Alle Kerne führen **parallel** denselben Code aus
- Variablen können sich aber unterscheiden
- Beispiel: gleichzeitig mehrere Teile eines Arrays verarbeiten



http://www.icp.uni-stuttgart.de

OpenMP – Beispiel

```
#include <stdio.h>
#include <omp.h>
int main() {
  #pragma omp parallel
    printf("thread %d\n", omp_get_thread_num());
  printf("fertig\n");
  return 0;
}
```

- Benötigt Headerdatei „omp.h“ und Compilerflag „-fopenmp“ (gcc)
- `#pragma omp parallel` leitet eine parallele Region ein
- Die parallele Region endet mit dem Block oder Befehl
- Im Beispiel:
  - „thread x“ wird so oft ausgegeben, wie Kerne vorhanden sind
  - „fertig“ wird nur einmal ausgegeben

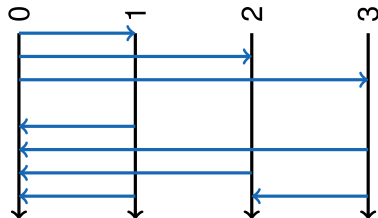
OpenMP – parallel for

```
  float a[100], b[100], result = 0;
  // parallele Initialisierung der beiden Vektoren
#pragma omp parallel for
  for (int i = 0; i < 100; ++i)
    a[i] = i; b[i] = 100 - i;
  // paralleles Skalarprodukt
#pragma omp parallel for reduction(+:result)
  for (int i = 0; i < 100; i++)
    result = result + (a[i] * b[i]);
```

- `#pragma omp parallel for` lässt OpenMP die folgende for-Schleife automatisch parallelisieren
- `reduction(+:var)` bedeutet, dass die Variable `var` über alle Threads aufsummiert wird
- man kann die Verteilung auf die Threads auch manuell über `schedule` beeinflussen

MPI

JUGENE, Jülich: 72 Racks × 1024 Nodes × 4 Kerne = 294912 Kerne



- Supercomputer verbinden viele Computer (Nodes) mit mehreren Prozessoren und Kernen über schnelle Netzwerke
- Dabei ist der Speicher nur lokal von den Knoten ansprechbar – OpenMP funktioniert so nicht
- Hier wird das Message Passing Interface (MPI) benutzt
- Erfordert wiederum andere Algorithmen als OpenMP

MPI – Beispiel

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char **argv) {
  int this_node;
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &this_node);
  printf("thread %d\n", this_node);
  if (this_node == 0) printf("fertig\n");
  MPI_Finalize();
  return 0;
}
```

- Bei MPI starten alle Threads wie ein normales Programm
- Soll Code seriell ausgeführt werden, müssen alle anderen Threads aktiv warten

MPI send/receive

```
MPI_Status status;
int empfangen[3], senden[3];
if ((this_node % 2) == runde) {
  MPI_Recv(empfangen, 3, MPI_INT, (this_node-1) % n_nodes,
    123, MPI_COMM_WORLD, &status);
} else {
  MPI_Send(senden, 3, MPI_INT, (this_node+1) % n_nodes,
    123, MPI_COMM_WORLD);
}
```

- Nachrichten sind Daten, die an andere Knoten geschickt werden
- Der Programmierer muss dafür sorgen, dass die Knoten wissen, wer wann wieviel an sie sendet

Runde 1
Node 0 Node 1 Node 2 Node 3
Runde 2

http://www.icp.uni-stuttgart.de