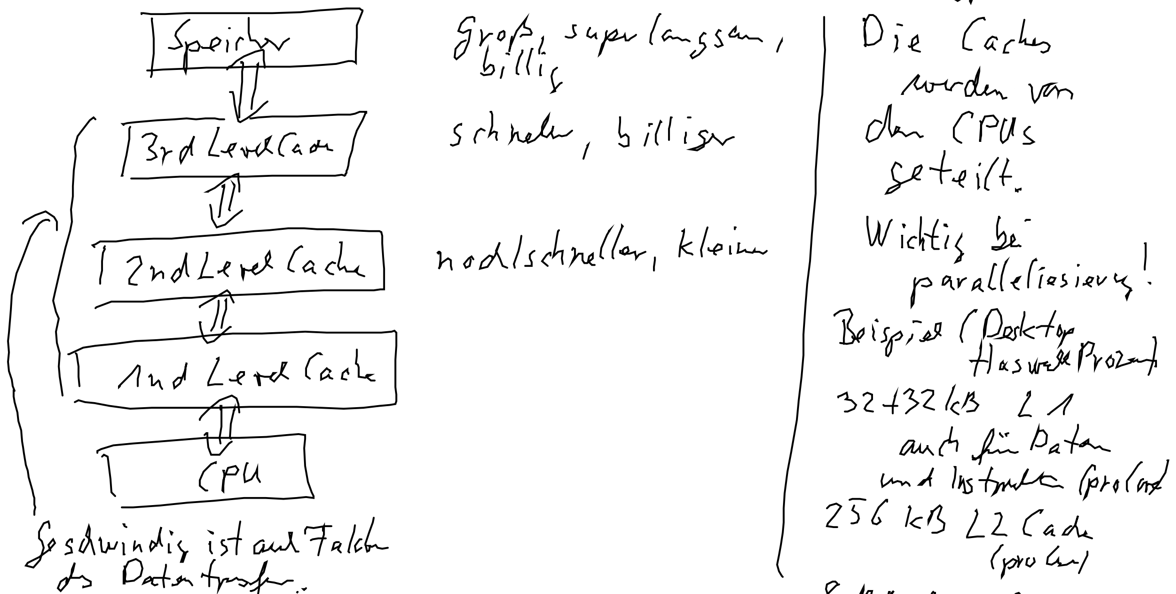
Fortsetzung Computerarchitektur

Kurzzusammenfassung letzte VL:

Ein Prozessor (CPU) hat viele Kerne (Cores)
Die CPUs haben verschiedene Einheiten (Fließkomma, Integer-, SIMD) Einheiten! Jede Instruktion benötigt unterschiedlich viele Takte.

Speicher sehr langsam

Um das zu kompensieren gibt Caches (Zwischenspeicher)



zum Vergleich Serverprozessor Xeon (variiert stark je Typ und Preis),
10 Kerne, 25 MB L3 Cache

Konsequenzen für Programmierer

1.) Beliebte Optimierungstaktik: Tabellierung
⇒ 1) kleine Tabellen anlegen, die möglichst ganz in den Cache passt.

1) Lieber viele kleine Tabellen, die sich mit wenigen schnellen Rechenoperationen zusammensetzen lassen können, als eine große Tabelle:

Bsp: nicht $e^{h[i]+g[j]}$ tabellieren,

sondern $e^{h[i]}$ und $e^{g[j]}$

2.) Speicherzugriffe: Möglichst zusammenhängende Bereiche, die in den Cache passen beschreiben.

3.) Passen die Daten auf die hintereinander zugegriffen wird nicht in den Cache ist die Speicherbandbreite entscheidend

(Die Bussen limitieren die Speicherbandbreite pro Rechner.) => viele Rechner.

<u>Organisation des Zwischenspeich</u>

Die Daten für den Cache werden nicht byteweise abgerufen, sondern in festen Abschnitten. Dies entspricht oft den Seiten in der Speicherorganisation.

(Es gibt virtuelle (durch Betriebssystem verwaltet) und physische Seiten) (z.B. Haswell 4 kB Seiten!)

<u>Optimierungsziel:</u> Verwende in einem Codeabschnitt möglichst wenige Seiten!

(Seitengröße nachschlagen oder ausprobieren?)

Das könnte aufgrund des Caches auch größere Einheiten sein. => z.B. 1MB?

wichtig nachlesen und ausprobieren und messen.

<u>Beispiel</u> 2D Array:

a[y * length + x]

Speicher (x,y)

| (1,0), (2,0), (3,0), ..., (length,0), (1,1), (2,1), (3,1) ...,
(length,1) ....)

Also length sei groß!

```
for( x=0 ; x < length ; x++) {
  for( y=0 ; y < length ; y++) {
     Mache etwas mit a[y×length+x];
  }
}
```

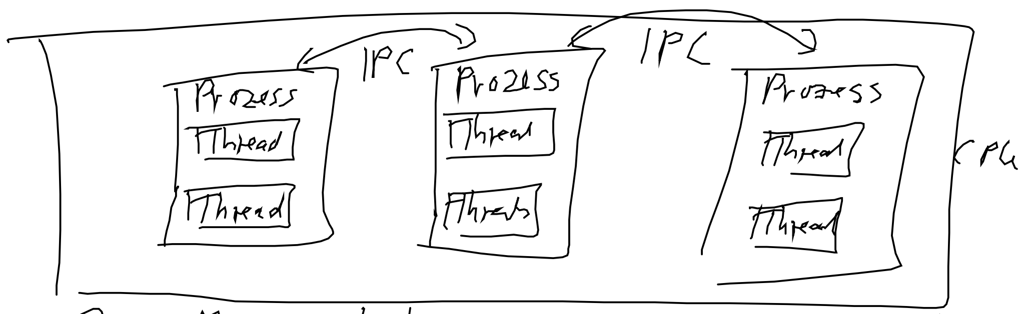
↑ Zugriff ist weit verstreut im Speicher.

langsamer als

```
for( y=0 ; y < length ; y++) {
  for( x=0 ; x < length ; x++) {
     Mache etwas mit a[y×length+x]
  }
}
```

Der Unterschied kann sehr groß sein (Größenordnung)
Generell kann ein (temporäres) Umpacken von Daten um das Memoryzugriffsmuster zu verbessern, einen enormen Geschwindigkeitsvorteil bringen! (Dazu später mehr)

Parallelisierung auf einer Maschine



Jeder Prozess hat sein eigenen virtuellen Addressraum (Seiten Cache).

Ohne zusätzliche Mechanismen kann kein Prozess auf den Speicher des anderen Prozesses zugreifen (Segmentation Fault)

Threads sind im gleichen Prozess und können auf den ~~gleich~~ gleichen Speicher zugreifen.

Im Prinzip 3 Möglichkeiten zur Parallelisierung (auf einer Maschine)

1) Mehrere Threads in einem Prozess
(Kommunikation u.a. durch gemeinsamen Speicher (+ Mutex, Locks)

2) Mehrere Prozesse
(Kommunikation durch Interprozesskommunikation (manchmal geteilter Speicher, Pipes) oder auch Netzwerkinterfaces

3) Kombination von beiden!

Grundprobleme bei beiden Varianten
- Synchronisierung (bei multithreading führt zu Abstürzen bei Fehlern)
  Besonders problematisch ist das Warten auf andere Prozesse.

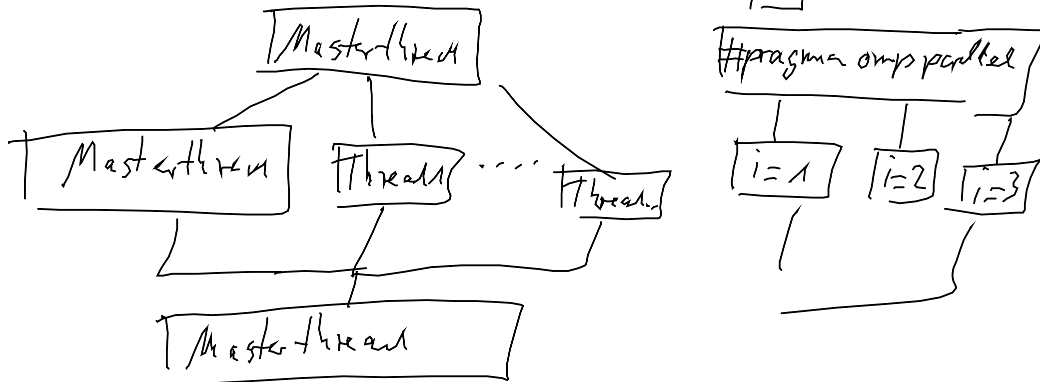
=> Ziel: Algorithmen sollten in ihren einzelnen Teilen möglichst unabhängig voneinander arbeiten.
Berechnung eines Thread/Prozess sollten möglichst wenig oder gar nicht von anderen Prozessen/Threads abhängen!

Realisierung Auswahl

multithreading (shared memory)
- per Hand (Linux, z.B. pthread), nicht zu empfehlen für wiss. Anwendungen, benötigt viel Handarbeit, braucht Erfahrung, nicht unbedingt praktikabel.

- OpenMP, Compilerdirektiven, die Schleifen automatisch parallelisieren, task direktiven (im Prinzip auch für Dummy geeignet, performance dann aber unklar) sehr leicht zu programmieren, wenig Aufwand.

  Fork-join Modell



Auch tw. SIMD im Hintergrund ...

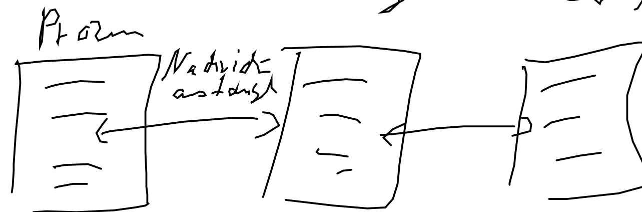
- OpenCL: Standard der die Maschine abstrahiert.
  ⇒ Prozessoren und Grafikkarten auf die gleiche Weise programmieren.
  Nachteil programmiert sich ähnlich wie OpenGL
  Sehr ungewohnter Zugang für viele Einstiegshürde.
- Bibliotheken die im Hintergrund parallelisiert sind
  z.B. ViennaCL, ...,

Wichtige Bemerkung: Bibliothek müssen threadsafe sein, damit die verwendet werden können!

Mehrere Prozesse

- per Hand schreiben, mit Interprocesskommunikation (z.B. Pipe) meist wenig praktikabel.

- MPI (Industriestandard): Standard Nachricht / Daten zu transferieren (geht auch über Netzwerk)



Definierte Punkt an dem etwas gesendet wird.
Wichtig, zwischen Senden und Empfangen, die Zeit nutzen und etwas sinnvolles tun.

- Library, die MPI verwendet
  Generell: Ausgereifte Numerikbibliotheken verwenden, die flexibel (wichtig gut bekannt)