

Fortsetzung Computerarchitektur

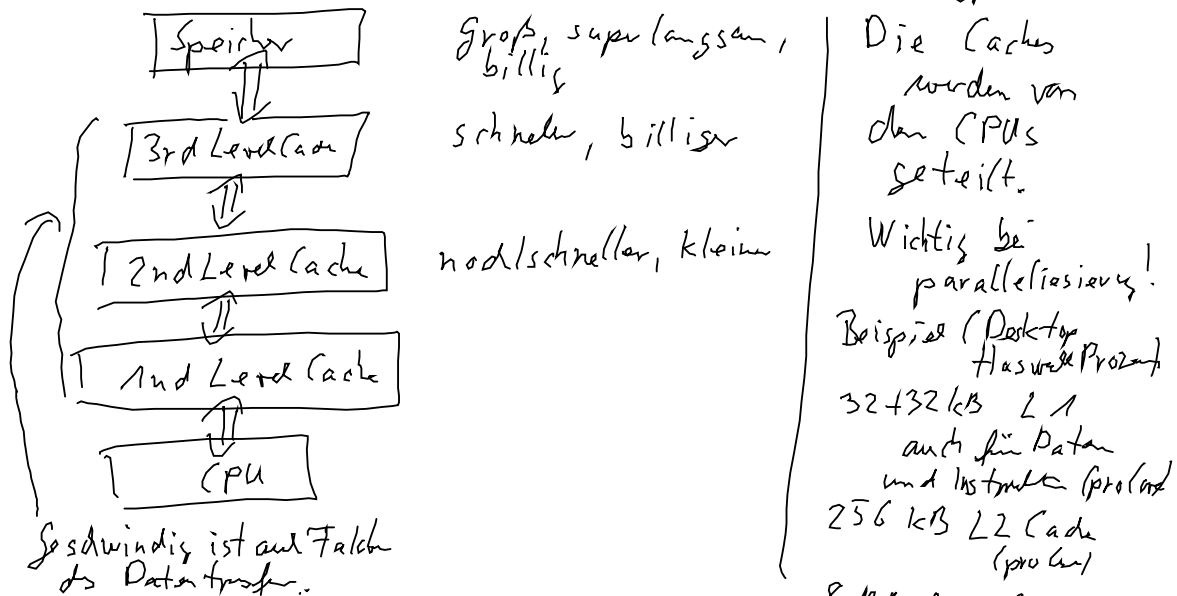
Kurz zusammenfassung letzte VL:

Ein Prozessor (CPU) hat viele Kerne (Cores)

Die CPUs haben verschiedene Einheiten (FPU, Komma, Integer, SIMD) Einheiten! Jede Instruktion benötigt unterschiedlich viele Takte.

Speicher sehr langsam

Um das zu kompensieren gibt Caches (Zwischspeicher)



Die Caches werden von den CPUs geteilt.
Wichtig bei parallelisierung!
Beispiel (Desktop Hauswerk Prozessor)
32+32 kB L1 auch für Daten und Instruktion (pro Core)
256 kB L2 Cache (pro Core)
8 MB L3 Cache (shared between CPU und GPU)

Zum Vergleich Serverprozessor Xeon (variant state je Typ und Proz.)
10 Kerne, 25 MB L3 Cache

Konsequenzen für Programmierer

- 1.) Beliebte Optimierungstechnik: Tabellierung
=> 1) kleine Tabellen anlegen, die möglichst ganz in den Cache passt.

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

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

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

3.) Passieren die Daten auf die hinterher nicht zugegriffen wird nicht in den Cache ist die Speicherbandbreite entscheidend

(Die Buslimitation der Speicherbandbreite pro Rechner.) \Rightarrow viele Rechner.

Organisation des Zwischenspeichers

Die Daten für den Cache werden nicht Byteweise abgefragt, sondern in festen Abschnitten. Dies entspricht oft der Seite in der Speicherorganisation.
(Es gibt virtuelle (durch Betriebssystem verwaltet) und physische Seiten) (z.B. Hauptspeicher 4 kB Seiten!)

Optimierungsziel: Verwend in ein Codeabschnitt möglichst wenige Seiten!
(Seitengröße nachschlagen oder ausprobieren?)

Das könnte aufgrund des Caches auch größere Einheiten sein. \Rightarrow z.B. 1 MB

wichtig nachher und ausprobieren und messen.

Beispiel 2D Array:

$$a[y * \text{length} + x]$$

Speicher (x, y)
 $(1, 0), (2, 0), (3, 0), \dots, (length, 0), (1, 1), (2, 1), (3, 1), \dots,$
 $(length, 1), \dots$

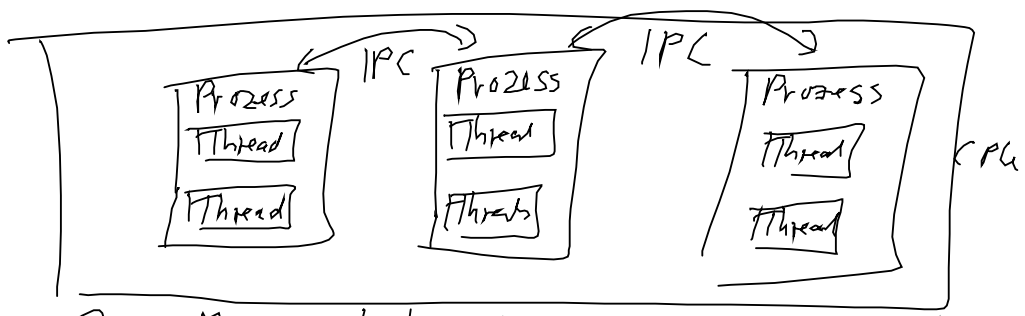
Also $length$ sei groß:
 for $(x=0; x < length; x++)$ {
 for $(y=0; y < length; y++)$ {
 Mache etwas mit $a[y \times length + x]$;
 }
}

↑
 Zugriff ist
 weit verstreut
 in Speicher.

langsame als
 for $(y=0; y < length; y++)$ {
 for $(x=0; x < length; x++)$ {
 Mache etwas mit $a[y \times length + x]$
 }
}

Der Unterschied kann sehr groß sein (Größenordnung)
 Gewisse kann ein (temporäres) Umpacken von Daten
 um das Memoryzugriffsmuster zu verbessern, ein engeres
 Geschwindigkeitsverteilung bringen! (Dazu später mehr)

Parallelisierung auf einer Maschine



Jeder Prozess hat seine eigenen virtuellen Adressen (Seite 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 ~~selben~~ 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 in Thread / Prozess sollten möglichst wenig oder gar nicht von anderen Prozessen ^{Threads} abhängen!

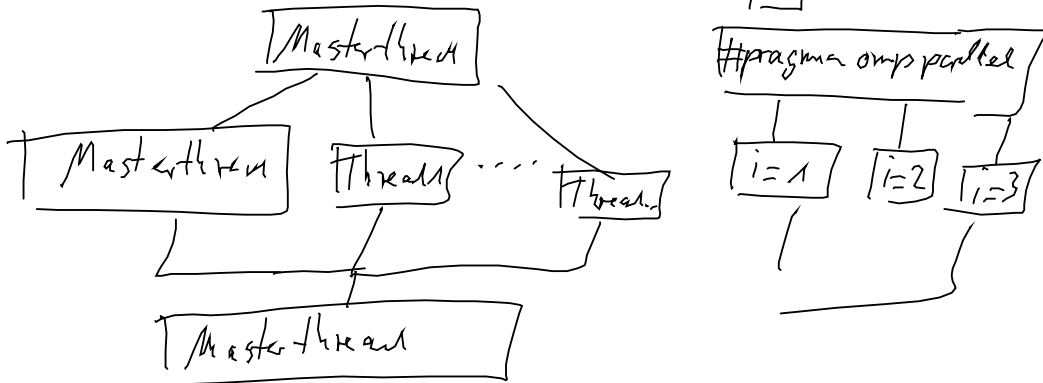
Realistischer Auswahl

multithreading (shared memory)

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

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

Fork-join Modell



Auch tw. SIMD im Hintergrund ...

- OpenCL: Standard der die Maschine abstrahiert.
=> Prozessen und Grafikkarten auf die gleiche Weise programmieren.

Nachteil programmieren ist ähnlich wie OpenCL

Sehr ungewohnter Zugang für viele Einstiegschürde.

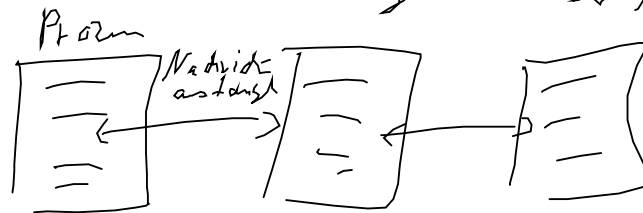
- Bibliotheken die im Hintergrund parallelisiert sind
z.B. ViennaCL, ...

Wichtig Bemerkung: Bibliothek müssen thread safe sein, damit die verwendet werden können!

Mehrere Prozesse

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

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



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

- Library, die MPI verwendet

Sourcecode: Ausgereifte Numerik Bibliothek verwenden,
die flexibel (richtig gut bekannt)