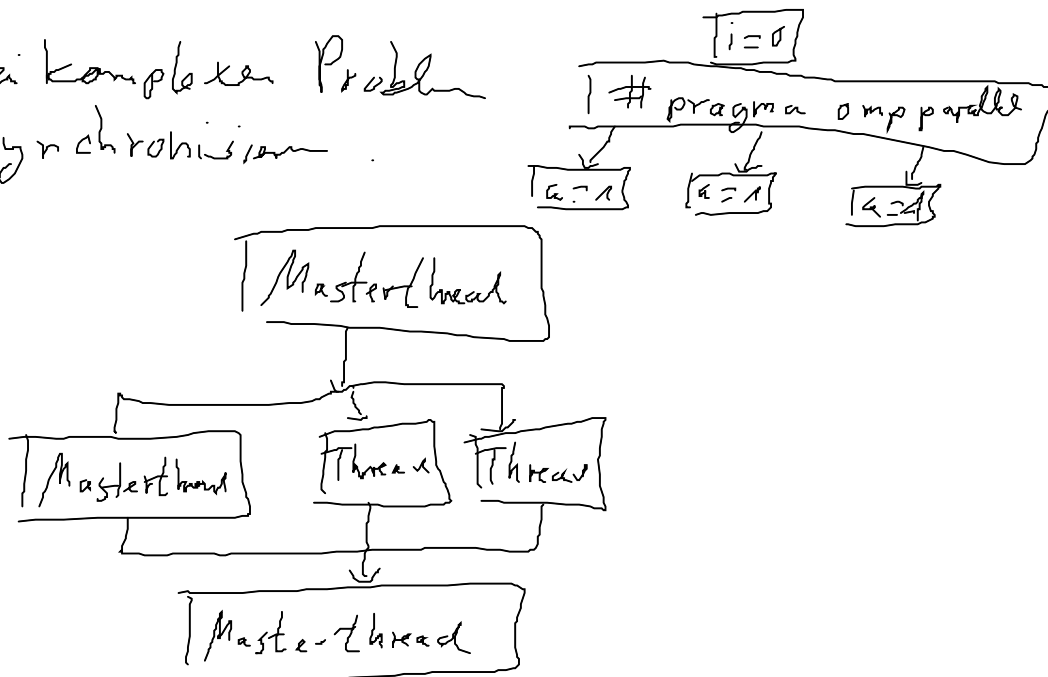


Realisierung von parallel laufenden Programmen

multithreading (shared memory)

- per Hand (Linux, z.B. pthread Library), nicht zu empfehlen für wiss. Anwendung, benötigt viel Handarbeit und viel Erfahrung...
- OpenMP, Compilerdirektiven, die Schleifen automatisch parallelisiert. (in Prinzip für Dummy geeignet, Performance nicht so gut ohne zusätzliche Optimierungen)
Sehr leicht zu Programmierung
Fork/Join Modell

Bei komplexen Problem
Synchronisation



Auch zw. SIMD im Hintergrund (complex)

- OpenCL: Standard der die Maschine abstrahieren
=> Prozessor und Grafikkarte auf die gleiche Weise programmieren
Nachteil programmiert sich wie OpenMP
Sehr ungewohnter Zugang für viele
Einstiegshürde

- Bibliotheken, die im Hintergrund parallel sind
z. B. Vienna CL, ...

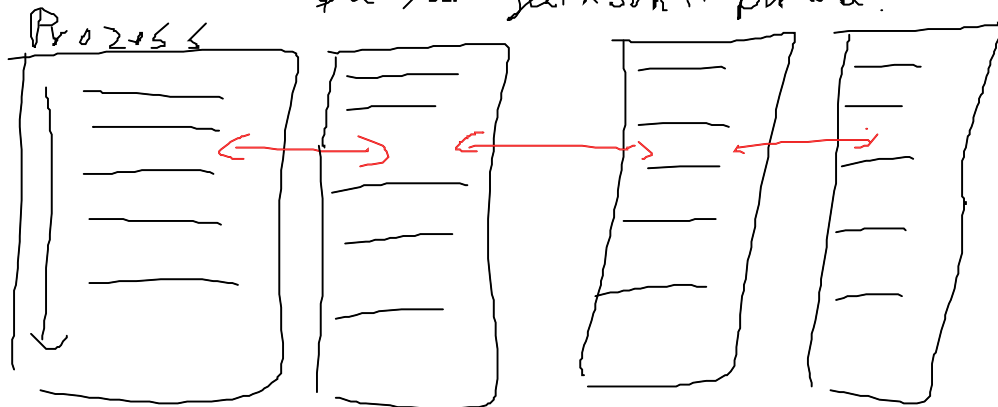
Wichtige Bemerkung: Bibliothek müssen thread safe sein, damit diese verwendet werden können.

Mehrere Prozesse

- portland schreiben, mit Interprozess (z.B. Pipes)
meist zu wenig portabel.
- MPI (Industriestandard), Standard um Nachricht/
Daten zu transferieren (z.B. Netzwerk)

Konzept zur Programmierung:

Prozesse laufen im Gleichschritt parallel:



Definierte Punkte an dem etwas gesendet oder empfangen wird.

Wichtig zur Senden und Empfangen etwas sinnvolls
zu tun.

- Library, die MPI verwendet

Genau: Ausgereifte numerische Bibliothek,
die flexibel verwendet werden können

GPU; Spezielle Hardware

Plattform: Wichtig insoweit Spezialhardware
für GPU Rechner für numerische Anwendungen.
oder Xeon Phi

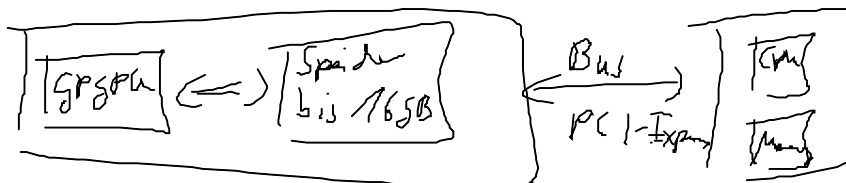
Hauptunterschied zur CPU

sehr viele Kerne, spezialisiert auf bestimmte Operationen
geringer Takt!

Beispiel: Pascal Architektur von Nvidia
Chip für Supercomputer (GP100)

56 x SM (multiprozessor): 32 double precision
und 64 single precision
(Bei Grafikarten 128 single precision
und 4 double precision)

Eine SM teilt sich ein L1 Cache für Texture (Array)
und 64 KB Shared Memory
Gemein L2 Cache 4096 KB



Transfer zur CPU langsam

Rechnung auf GPU liefert nur an Anfang und

Ende (oder selten) mit CPU interagieren (sonst langsam)

Wichtig Genauig der Fließkommaoperation größer als
CPU (Situation bei GPU Chips
bzw.)
(Gemeint andere Datentyp, tw. kein
Integer)

Wegen massiver Parallelität \Rightarrow Programmierung und
Befehle anders.

Viele Recheneinheiten führen die selben Operationen
aus auf großen Datenblöcken. (Anderer Art von Befehl)

Damit die Shader nicht aus dem Takt kommen
gibt es kein Sprünge (z.B. CPU)

CPU

```
if (Bedingung) {  
    Code mit vielen Zeilen  
}
```

Falls
Bedingung
unwahr ist
Sprung

GPU:

```
if (Bedingung) {  
    Code mit vielen Zeilen  
}
```

\leftarrow Falls Bedingung unwahr ist,
wird Code trotzdem ausgeführt,
aber er ändert nichts!

Vorteil Synchronisierung nicht so schwierig.

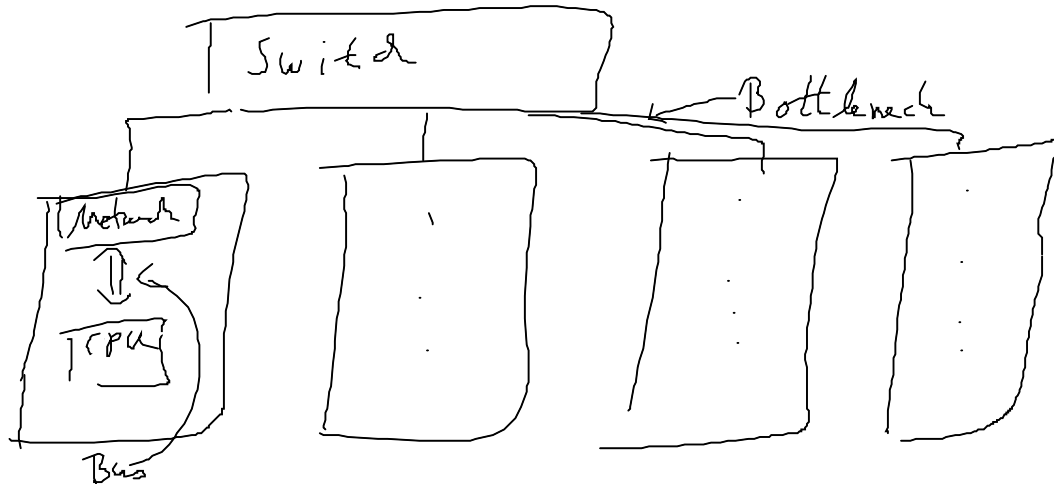
Auch hier sieht es Möglichkeit zur Synchronisierung,
aber auch gilt vermeiden.

Programmierung:

- Open GL (Eigentlich für Grafik,
nicht wirklich zu spielen)

- CUDA (nur nvidia support, ist aber weit verbreitet)
- OpenCL (Hersteller unabhängig)
oder Bibliothek verwenden.

Mehrere Rechner im Cluster



Bei größeren Anwendungen Kopplung verschiedener Rechner.
mehrere Möglichkeiten:

- 1 Prozess je Rechner mit
vielen Threads (OpenMP)
- mehrere Prozesse je Rechner
mit single Threads

Kommunikation in der Regel mit MPI
Bibliothek (verschiedene Implementierungen,
OpenMPI, mpich2)

Vernetzung: Herkömmlich Gigabit Ethernet, IP.
(Hardware zu Layer 2, Ethernet,
IP Protokolle zu höherer Schicht)

richtiger Cluster => oft: Infiniband (z.B. 20 Gbit
und aufwärts)

RDMA Protokolle
(Remote Direct Memory Access)

Direkter Transfer von Speicher zu Speicher
ohne viel Overhead.

(Wenn korrekt konfiguriert mit MPI
das im Hintergrund)

Frage: Knoten mit viel Cores und vielen Speichern
mehr Knoten mit wenig (CPU) und wenig Speicher

Vorteil: bei CPU-lastigen Anwendungen:
geringer Preis, geringere Anforderungen an
Skalierbarkeit

Höhere Speicherbandbreite möglich,
da Speicherbus mehr Busse
bzw. Transfer.

Widerr Vorreicht es um viel Daten über das
Netzwerk zu senden (langsam).