Computerorientierte Quantenmechanik

1. Einführung

Marten Richter, EW 7 10
Tel: 314-24858

Email: mrichter@itp.tu-berlin.de
Sprechstunde: n. V.

Thema: **Was es nicht ist:**

kein Programmierkurs und kein C-Kurs (Algorithm)

kein Quantencomputing

kein typisch theoretischer Kurs

Was es ist

Experiment: Verbindung der Praxis der computerorientierten Physik der Theoretik Physik – mit der Informatik

Bunte Mischung an anwendungsnaher Informatik, Mathematik auf verschiedenen Gebieten der Theoretik Physik.

Das Programm (vorläufig)

1.) Einführung

Informatik I
2.) Rechnerarchitekturen
3.) Lineare Algebra für parallel Programmieren

Physik II / Informatik / Mathematik

4.) Lösung der Schrödingergl. über Finite Differenzen f. Exzitonen und Trionen

5.) Heuristikverfahren: Lanzos Verfahren für Eigenwerte. Krylow-Schur Verfahren

6.) Preconditioning von Matrizen

Physik/Chemie III

7.) Basissysteme der Quantenchemie

8.) Configuration Interaction Methode

9.) Dichtefunktionaltheorie

10.) Born-Oppenheimer → Optimierungsproblem

Physik / Informatik IV

11.) Zeitliche Dynamik: Diabatenmatrixtheorie
Blockgleichung

12.) Beispiele für Zeitsolver

13.) Vielteilchendynamik, Korrelationsentwicklung, Coulombstörung

14.) Stationäre Probleme und Matrix Dynamik

V

15.) Fourier Transformation DFT / FFT

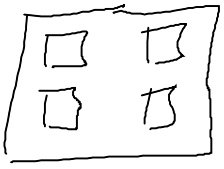
16.) Harmonische Inversion

I.2) Rechnerarchitekturen

Für effiziente Programmierung muss man

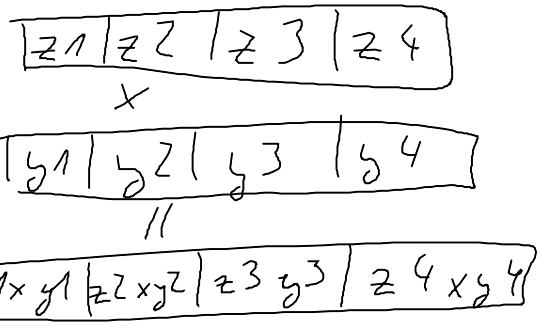
Wissen wie der Computer gebaut ist.
(Sehr vereinfachte Darstellung)

Die CPU



- Besteht aus mehren Kern (Core)
  Jeder Kern besteht aus mehren Rechenwerken
  - Fließkommaeinheiten
- SIMD (single input multiple data)
  (mehrere gleichartige Instruktionen auf einmal
  z.B bei Vektoroperationen z.B. MMX, SSE, SSE2 AVX-.)

z.B



optimierter Code für Vektoroperationen (schwierig, gute Compiler oder optimierte BLAS z.B f. lin Algebra)

- Integer Einheiten

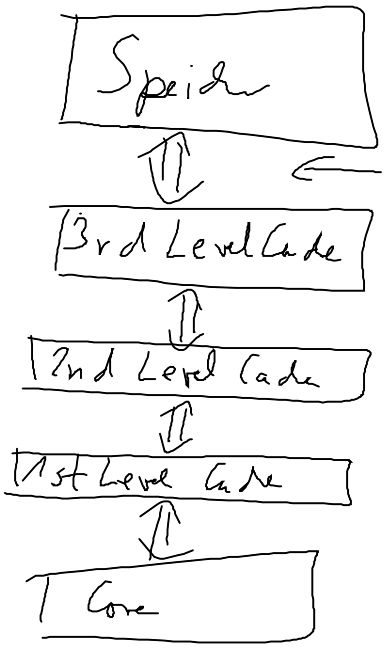
Bem: Manchmal teilen sich Cores Einheiten

Meist gibt es eine Einheit, die die Befehle dekodiert und auf die Rechenwerke verteilt. (Teilweise autoparallel)

(Bei Hyperthreading werden zwei Cores simuliert, um die Einheiten besser auszunutzen)

Wichtige Regel für Programme: Finde heraus, was am längsten im Programm dauert (z.B. Profiler)
Finde heraus, ob es die Rechenzeit ist.
Dann optimiere nur diese Teile!

Aber meistens ist es Speicher ..... , der Speicherzugriff ist langsam.



langsam

Speicherbus (langsam)

schnell, billig (groß)

noch schneller, teurer, kleiner

super schnell, super teuer, sehr klein

Die Caches werden teilweise zwischen den Cores geteilt

Beispiel (Desktop Haswell Prozessor)

32 kB 32 kB L1 Cache für Daten und Instruktionen (pro Core)

256 kB L2 Cache (pro Core)

8 MB L3 Cache (geteilt)

Zum Vergleich Serverprozessor Xeon (privat extrem Preis), 10 Core 25 MB L3 Cache

Konsequenzen für die Programmierung:

1.) Beliebte Optimierungstechnik: Tabellierung

=> 1) Kleine Tabellen anlegen, die möglichst im niedrigen Cachelevel passen

2) Lieber viele kleine Tabellen, die mit wenigen Rechenoperationen ausgerechnet werden können als eine große Tabelle.

Beispiel nicht `h[i]+g[j]` tabellieren, sondern `h[i]` und `g[j]`

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

3.) Passen die Daten nicht in den Cache
ist die Speicherbandbreite entscheidend
(Bei Speicherbandbreite am Limit => viele Rechner)

Organisation des Zwischengespeicherten im Cache:

Die Daten werden nicht Byte für Byte abgelegt,
sondern in festen Abschnitten. Dies entspricht
oft den Seiten in der Speicherorganisation
(Es gibt virtuelle (durch Betriebssystem verwaltet)
und physikalische) (z.B. Haswell 4 KB Seiten!)

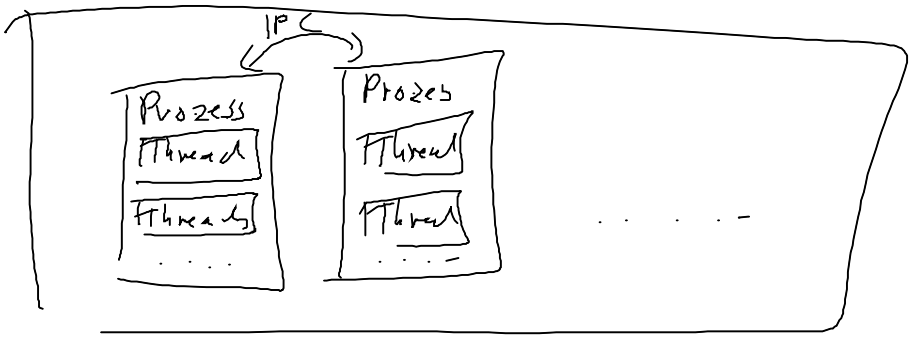
Optimierungsziel Verweilen im Code absolute
möglichst wenige Seiten!
Das kann auch große Einheiten sein z.B 1 MB!
=> Wichtig ausprobieren und messen!

Beispiel 2D Array $a[y \times \text{length} + x]$
Reihenfolge der Schleife

Der Unterschied kann sehr groß sein (Speicherbedarf
Umpacken (temporär) kann sich (ohne)

Parallelisierung auf einer Maschine

Programmausführung auf einer Maschine



Jeder Prozess hat seinen eigenen virtuellen Adressraum.
Ohne explizite Mechanismen kann ein Prozess
nicht auf den Speicher eines anderen Prozesses zugreifen.
(Segmentation Fault (SIGSEGV))

Threads sind innerhalb eines Prozesses und können
auf denselben Speicher zugreifen.

Im Prinzip 3 Möglichkeiten zur Parallelisierung
(auf einer Maschine)

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

2) Mehrere Prozesse
(Kommunikation durch Interprozesskommunikation
(geteilter Speicher, Pipes ...) oder Netzwerkinterface ....

3) Mischung aus beidem

Grundprobleme bei beiden Varianten

- Synchronisierung

Besonders problematisch ist das Warten

⟹ Ziel: Algorithmen verwenden, die Teilprobleme
bearbeiten, die kaum von einander abhängen.

Berechnung von Threads/Prozessen sollte möglichst
wenig oder gar nicht von den anderen Threads/Prozessen
abhängen.