Parallele Simulation

Dr.-Ing. Volkmar Sieh

Department Informatik 4
Systemsoftware
Friedrich-Alexander-Universität Erlangen-Nürnberg

WS 2025/2026

Parallele Simulation

Typischer Desktop-PC hat z.Z. Quad-Core-CPU.

Gute Nutzung der Cores könnte Virtuelle Maschine um Faktor 4 beschleunigen...

Threads

Zwei Möglichkeiten:

- Es existiert ein Pool von (z.B. 4) Threads, die sich jeweils aus einem Pool von Aufträgen (auszuführende „step"-Funktionen der einzelnen Komponenten) Aufträge herausnehmen und diese abarbeiten.
  Nach der Abarbeitung aller Aufträge wird die Zeit erhöht.
  Danach werden wieder alle „step"-Funktionen einmal aufgerufen, usw.
- ...

Problem: Threads müssen sich nach der Abarbeitung der Aufträge jeweils synchronisieren (=> Performance-Problem)

Threads

- ...
- Für jede Komponente existiert genau ein Thread der die „step”-Funktion der Komponente immer wieder aufruft.
  Die Uhr ist auch eine Komponente und schaltet in ihrer „step”-Funktion mit Hilfe ihres Threads die Uhr weiter.

Probleme:

- Komponenten können zeitlich „auseinander driften” aufgrund unterschiedlicher Verdrängungen der einzelnen Threads.
- Relative Performance der Komponenten schwierig einzuhalten.

Parallele Simulation

Parallelisierung auf verschiedene Ebenen denkbar:

- jeder PC hat eigenen Thread
- ...
- jede Komponente hat eigenen Thread
- ...
- jeder Chip hat eigenen Thread
- ...
- (jedes Gatter hat eigenen Thread)

Ein Thread pro PC

Im folgenden: jeder Thread simuliert einen PC

Vorteile:

- Koordinierung/Synchronisierung der echten Cores nur selten notwendig

Nachteile:

- ggf. nur wenige echte Cores nutzbar

Problem: kritische Abschnitte

Angenommen, PC 1 sendet Netzwerk-Nachricht zu PC 2.
Dann führt Thread von PC 1 (grob) folgende Funktionen aus:

Innerhalb von PC 1:

- CPU-Simulation
- Bus-Write-Funktion
- Callback-Funktion in Sender-Netzwerkkarte
- Callback-Funktion im Netzwerk-Kabel

Netzwerk-Kabel-Simulation transferiert Kontrollfluss zu PC 2(!)...

Problem: kritische Abschnitte

... und in PC 2 werden folgende Funktionen ausgeführt:

- Callback-Funktion in Empfänger-Netzwerkkarte
  (schreibt per DMA Speicher, setzt Interrupt im Status-Register
  => kritischer Abschnitt!)
- Callback-Funktion im PIC
  (setzt IRR, ISR, ...
  => kritischer Abschnitt!)
- Callback-Funktion in CPU
  (setzt Interrupt-Pending-Flag
  => kritischer Abschnitt!)

=> viele kritische Abschnitte

Ein Thread pro Komponente

Im folgenden: jeder Thread simuliert eine Komponente

Problem:
noch mehr kritische Abschnitte; Beispiele

- Platte führt gerade Auftrag aus; bekommt parallel neuen Auftrag
- Grafikkarte liest gerade Speicher aus; Speicher wird parallel beschrieben
- Uhr tickt gerade; CPU liest Zeit parallel aus
- zwei CPUs führen gleichzeitig ein `cmpxchg` aus
- ...

Ein Thread pro Komponente

kritische Abschnitte entsprechen den „Problem"-Zonen in echter Hardware

Beispiele:

- das Status-Register einer echten seriellen Schnittstelle muss ausgelesen werden können, obwohl sich gleichzeitig der Status ändert
- wenn der PIC von der CPU gefragt wird, welchen Interrupt er gerade signalisiert, muss er zum gleichen Zeitpunkt einen weiteren Interrupt entgegennehmen können
- zwei Cores müssen serialisiert werden, wenn sie zum gleichen Takt-Zeitpunkt auf den Bus zugreifen wollen
- ...

Problem Buszyklen

In Wirklichkeit werden CPU-Cores über den Bus synchronisiert.

Beispiel x86: Lock-Prefix (z.B. `lock addl $1, counter`)

Eigentlich:

```
bus_lock(bus);
tmp0 = load(counter);
tmp1 = 1;
tmp2 = add(tmp0, tmp1);
store(counter, tmp2);
bus_unlock(bus);
```

Bisher nur:

```
tmp0 = load(counter);
tmp1 = 1;
tmp2 = add(tmp0, tmp1);
store(counter, tmp2);
```

Problem: ein simuliertes Bus-Locking (Spinlock, Mutex, ...) dauert viel zu lang.

Problem Buszyklen

Problem: ein echtes Bus-Locking (Spinlock, Mutex, ...) dauert viel zu lang.

Idee(?): `bus_lock`- und `bus_unlock`-Aufrufe nur bei Lock-Prefix.

Problem (Beispiel):

```
spinlock_lock:
1:   movl $0, %eax
     movl $1, %edx
     lock cmpxchgl %edx, lock
     jne 1
     ret
```

```
spinlock_unlock:
     movl $0, lock
     ret
```

Zugriff nur zum Teil mit Lock-Prefix!

Problem Buszyklen

Idee: Mit Lock-Prefix versehene Instruktionen mit `cmpxchg` nachbilden.

`addl $1, counter`:

```
tmp0 = load(counter);
tmp1 = 1;
tmp2 = add(tmp0, tmp1);
store(counter, tmp2);
```

`lock addl $1, counter`:

```
do {
  tmp0 = load(counter);
  tmp1 = 1;
  tmp2 = add(tmp0, tmp1);
} while (! cmpxchg(counter, tmp0, tmp2
```

Hinweis: Idee funktioniert nur mit direkt gemapptem Speicher!

Problem JIT

JIT legt compilierten Code in einem CPU-Simulations-internen Cache ab.

Angenommen, zu simulieren sei eine Quad-Core-CPU.

Zwei Möglichkeiten:

- jeder Core hat eigenen Code-Cache
- alle Cores haben gemeinsamen Code-Cache

Probleme:

- Ggf. Nebenläufigkeit beim Code-Eintragen bzw. -Invalidieren.
- Pages mit compilierte Blöcken müssen in allen TLBs als „geschützt" eingetragen werden.
- Was passiert, wenn ein Core gerade einen Block compiliert, den eine andere gerade im Speicher überschreibt?
- ...

Problem Synchronisierung

Existieren für alle Komponenten Threads (=> viele Threads), muss das Host-OS häufig Kontext-Wechsel durchführen (nicht genügend echte Cores vorhanden).

Daher maximal nur soviele Threads (Cores) verwenden, wie die Host-Hardware besitzt.

Algorithmus-Idee: jeder Thread führt folgende Schleife aus:

```
while (true) {
  while (step_func_pool_not_empty()) {
    func = remove_step_func_from_pool();
    execute(func);
  }
  last = barrier();
  if (last) {
    increase_time();
  }
}
```

Problem Synchronisierung

Synchronisierung schwierig:

- Synchronsierung i.A. ein System-Call (teuer)
- ggf. ein Thread z.Z. verdrängt (=> alle anderen warten)

(2. Punkt z.Z. Forschungsgegenstand)