Parallelität durch Abstraktion

Seminar - Multicore Programmierung - WS 2010

Andreas Wilhelm

Fakultät für Informatik und Mathematik
Universität Passau

11. November 2010

Inhaltsverzeichnis

Synchronisation in Java (1)

Beispiel

```
MyObject obj = null;

synchronized( obj ) {

   // ...

}
```

Synchronisation in Java (1)

Beispiel

```
MyObject obj = null;

synchronized( obj ) {

   // ...

}
```

Problem

NullPointerException

Synchronisation in Java (2)

Beispiel

```
MyObject obj = new MyObject();

synchronized( obj ) {

   obj = new MyObject();

}
```

Synchronisation in Java (2)

Beispiel

```
MyObject obj = new MyObject();

synchronized( obj ) {

   obj = new MyObject();

}
```

Problem

Synchronisation auf anderem Objekt

Synchronisation in Java (3)

Beispiel

```
private static String lock = "lock";

synchronized( lock ) {

   // ...

}
```

Synchronisation in Java (3)

Beispiel

```
private static String lock = "lock";

synchronized( lock ) {

   // ...

}
```

Problem

Sperre von Strings mit gleichem Wert

Synchronisation in Java (4)

Beispiel

```
private static Integer var = 0;

synchronized( var ) {

   // ...

}
```

Synchronisation in Java (4)

Beispiel

```
private static Integer var = 0;

synchronized( var ) {

   // ...

}
```

Problem

Sperre von “autoboxed” Objekten mit gleichem Wert

Synchronisation in Java (5)

Beispiel

```
...
  public void addObserver(Observer<E> observer) {
    synchronized(observers) {
      observers.add(observer);
    }
  }

  public boolean removeObserver(Observer<E> observer) {
    synchronized(observers) {
      return observers.remove(observer);
    }
  }
...
```

Synchronisation in Java (5)

Beispiel

```
...
  private void notifyElementAdded(E element) {
    synchronized(observers) {
      for (Observer<E> observer : observers)
        observer.added(this, element);
    }
  }

  @Override public boolean add(E element) {
    boolean added = super.add(element);
    if (added)
      notifyElementAdded(element);
    return added;
  }
...
```

Synchronisation in Java (5)

Beispiel

```
public static void main(String[] args) {
  ObservableList<Integer> list = new ...

  list.addObserver(new Observer<Integer>() {
    public void added(Observable<Integer> s, Integer e) {
      System.out.println(e);
    }
  });

  for (int i = 0; i < 100; i++)
    list.add(i);
}
```

Synchronisation in Java (5)

Beispiel

```
list.addObserver(new Observer<Integer>() {
  public void added(Observable<Integer> s, Integer e) {
    System.out.println(e);

    if (e == 23) s.removeObserver(this);

  }
});
```

Synchronisation in Java (5)

Beispiel

```
list.addObserver(new Observer<Integer>() {
  public void added(Observable<Integer> s, Integer e) {
    System.out.println(e);

    if (e == 23) s.removeObserver(this);

  }
});
```

Problem

ConcurrentModificationException

Synchronisation in Java (5)

Beispiel

```
...
        if (e == 23){
          ExecutorService executor =
            Executors.newSingleThreadExecutor();
          final Observer<Integer> observer = this;
          executor.submit(new Runnable() {
            public void run() {
              s.removeObserver(observer);
            }
          });
...
```

Synchronisation in Java (5)

Beispiel

```
...
        if (e == 23){
          ExecutorService executor =
            Executors.newSingleThreadExecutor();
          final Observer<Integer> observer = this;
          executor.submit(new Runnable() {
            public void run() {
              s.removeObserver(observer);
            }
          });
...
```

Problem

Deadlock!

Warum Abstraktion von Parallelisierung?

- Reduzierung von Komplexität
- Steigerung der Effektivität
- Erhöhung von Portabilität
- Bessere Skalierbarkeit
- Forschung “steuern”

Domänenspezifische Umgebungen - Eigenschaften

Eigenschaften:

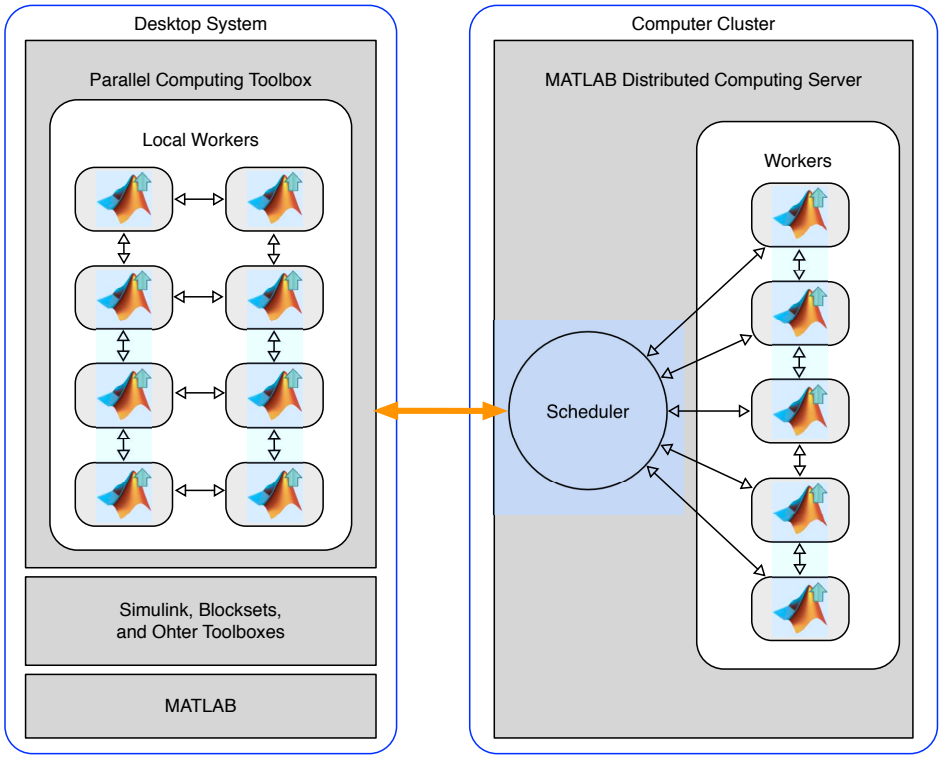
- Beinhalten maßgeschneiderte Notationen und Konstrukte
- Hohe Ausdrucksstärke
- Hohe Produktivität

Domänenspezifische Umgebungen - Beispiele

Beispiele:

- Apple's *CoreAudio*
- Google's *MapReduce*
- MATLAB®

Domänenspezifische Umgebungen - MATLAB (1)



- implizite sowie explizite Nebenläufigkeit
- lokal bis zu 8 *Worker*-Objekte
- Berchnung auf Cluster möglich (Scheduler)

Abbildung: *Parallel Computing Toolbox*

Domänenspezifische Umgebungen - MATLAB (2)

Worker-Methode:

1. Eine Anzahl an (lokalen) Workern anfordern (max. 8)
2. Die gewünschte MATLAB-Funktion wie gewohnt aufrufen. Bei Bedarf werden Aufgaben auf die verfügbaren Worker verteilt
3. Freigeben der Worker

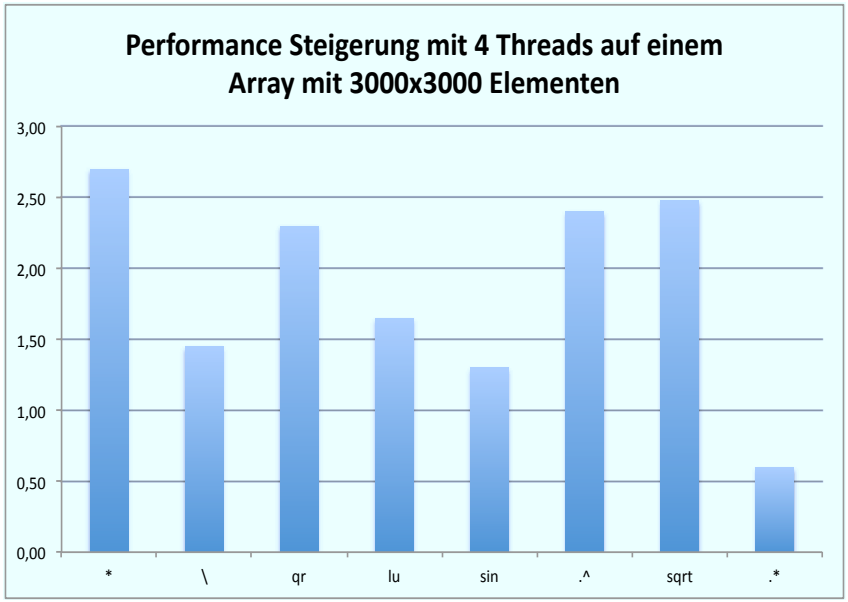
Beispiel

```
matlabpool open local 4
  calculate
matlabpool close
```

Domänenspezifische Umgebungen - MATLAB (3)

SPMD-Methode:

1. *Parallel Processing Toolbox* beinhaltet über 150 parallele Operatoren für die Verwendung auf Arrays, Matrizen, usw.
2. Neue Operationen lassen sich per `spmd`-Schlüsselwort selbst definieren



Domänenspezifische Umgebungen - Fazit

Vorteile

- Hohe Abstraktion von Parallelität
- Großes Optimierungspotential durch Fachwissen
- Aufteilung der Komplexität auf verschiedene Entwicklergruppen

Nachteile

- Boyle's Gesetz bzgl. DSLs: *Ausdrucksstärke * Domänengröße = Konstant*

Entwurfsmuster - Charakterisierung

Charakterisierung von Entwurfsmustern (Erich Gamma)

"Die Idee hinter Entwurfsmustern ist es, die Wiederverwendbarkeit bewährter Lösungen von Experten zu fördern."

Entwurfsmuster beinhalten i.A. vier informale Elemente:

1. Der Name des Entwurfsmuster
2. Die Problembeschreibung
3. Die Problemlösung
4. Die Konsequenzen

Entwurfsmuster - Bibliothek von UPCRC/Illinois

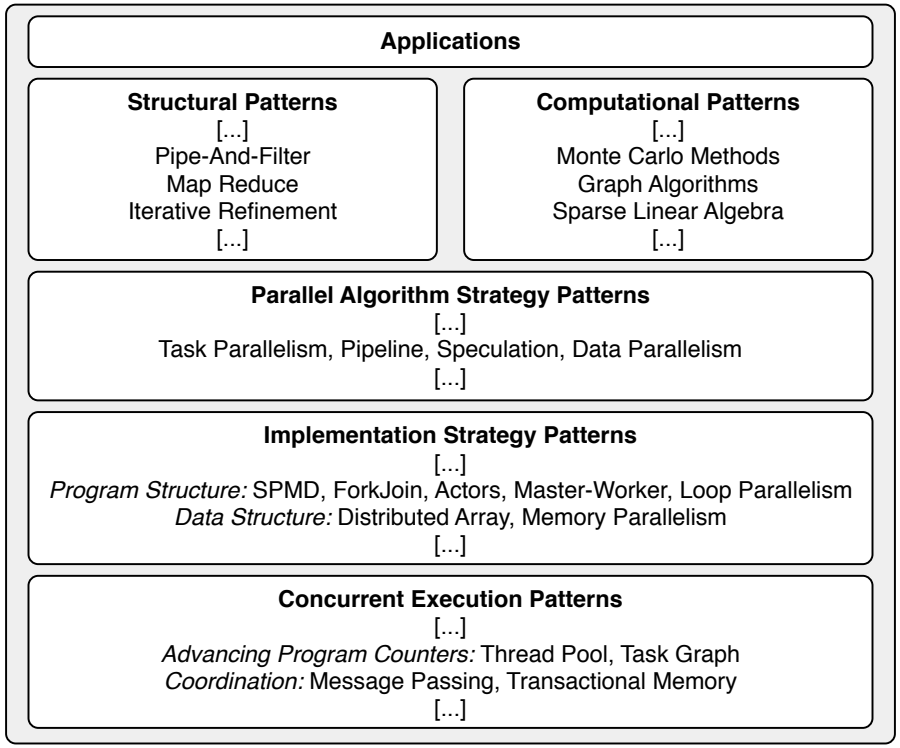


Abbildung: UPCRC/Illinois - Entwurfsmuster-Bibliothek[1]

[1] http://www.upcrc.illinois.edu/patterns.html

Entwurfsmuster - Beispiel

Beispiel

Pipeline-Entwurfsmuster, siehe Aktoren...

Entwurfsmuster - Fazit

Vorteile

- Verwendung bewährter Lösungen
- Führt zu abstrakterer Denkweise
- Verfügbare Beispiel-Implementierungen

Nachteile

- Informale Beschreibung
- Schwierige Auswahl der passenden Muster
- Abhängigkeit von Entwicklungsumgebung

Parallele Skelette - Erläuterung

Parallele Skelette:

- Parallele Programmierschablonen in Form von algorithmischen Mustern
- Beinhalten Plattform-spezifische (parallele) Implementierung
- Akzeptieren anwendungsspezifische Daten oder Funktionen als Parameter
- Mit Hilfe der Skelette können Anwendungen komponentenweise parallelisiert werden

Parallele Skelette - Einteilung

Kategorien von parallelen Skeletten:

- *Datenparallel*
  Ermöglichen Parallelisierung von Berechnungen auf strukturierten Daten
- *Taskparallel*
  Verschiedene Muster, um nebenläufige Aufgaben zu organisieren

Einschub: *Scala*-Programmiersprache (1)

Eigenschaften:

- Basiert auf der *Java Virtual Machine* (JVM)
- Beinhaltet objektorientierte, sowie funktionale Sprachelemente
- Jeder Wert ist ein Objekt
- Statisch typisiert
- Funktionale Ausdrücke werden strikt ausgewertet(verzögerte Auswertung durch Annotationen)
- Unterstützt Aktoren durch eigene Bibliothek (folgt später)

Einschub: *Scala*-Programmiersprache (2)

Syntax:

- Typangabe:
  `variable: typ`
- Deklaration von Werten:
  `val wert: Int = 42`
- Deklaration von Variablen:
  `var wert: Double = 1.32`
- Deklaration von Methoden:
  `def meth(par1: String, par2: Int): Unit`
- Definition von Funktionen:
  `function: [Typ] => [Typ]`

Einschub: *Scala*-Programmiersprache (3)

Futures:

- Ein auf Aktoren (folgt später) basierendes Konzept, das Platzhalter-Objekte für parallel berechnete Werte bietet
- Einzelne Berechnungen lassen sich asynchron in Threads ausführen
- Syntax: `val worker = future(5*42)`
- Ergebnisse lassen sich über `def awaitAll(timeout: Long, fts: Future[Any]*)` "einsammeln"

Parallele Skelette - Beispiel (1)

$$map\ f\ [a_1, \ldots, a_m] = [f\ a_1, \ldots, f\ a_m] \tag{1}$$

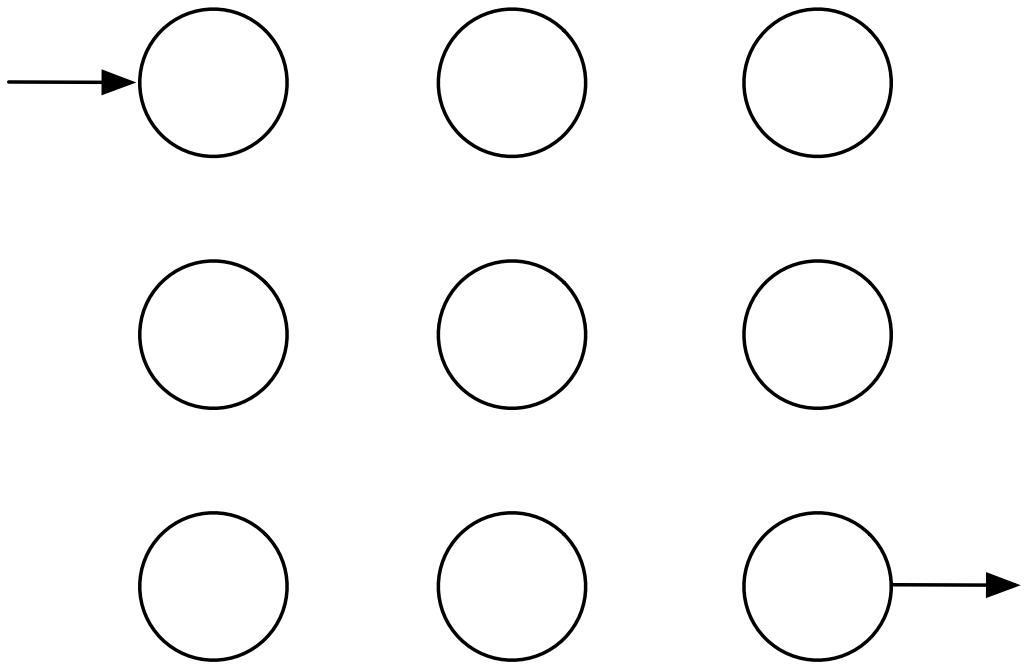


Abbildung: *map*-Skelett

Parallele Skelette - Beispiel (2)

Beispiel

```
val aMinute = 1000 * 60

def map[A, B](f: (A) => B)(list: List[A]): List[Any] = {
  val futures: List[Future[Any]] = list map{
    x => future( f(x) )
  }
  awaitAll(aMinute, futures: _*)
}
```

Aufruf

```
mapP[Int, Int](x => 2*x)(List(1,2,3,4,5))
```

Parallele Skelette - Fazit

Vorteile

- Wiederverwendbarkeit algorithmischer Muster
- Effektive Abstraktion von Parallelisierung
- Formale Beschreibung von Lösungen
- Es existieren effiziente Implementierungen
- Kombinationen von Skeletten möglich
- Mögliche Aufteilung in Implementierer und Verwender der Skelette

Nachteile

- Höchste Effizienz meist nur mit geeigneter Kombination möglich
- Verfügbarkeit paralleler Skelette
- Nicht für alle Anwendungsbereiche einsetzbar

Aktoren - Erläuterung (1)

Definition

Aktoren sind reaktive Entitäten mit eindeutigem Namen, welche ausschließlich per Nachrichten an ihre Eingangs-Mailbox angesprochen werden. Sie können:

1. Eine endliche Menge an Nachrichten an andere bekannte Aktoren senden.
2. Eine endliche Anzahl an neuen Aktoren erstellen.
3. Definieren, wie auf eingehende reagiert werden soll.

Aktoren - Erläuterung (2)

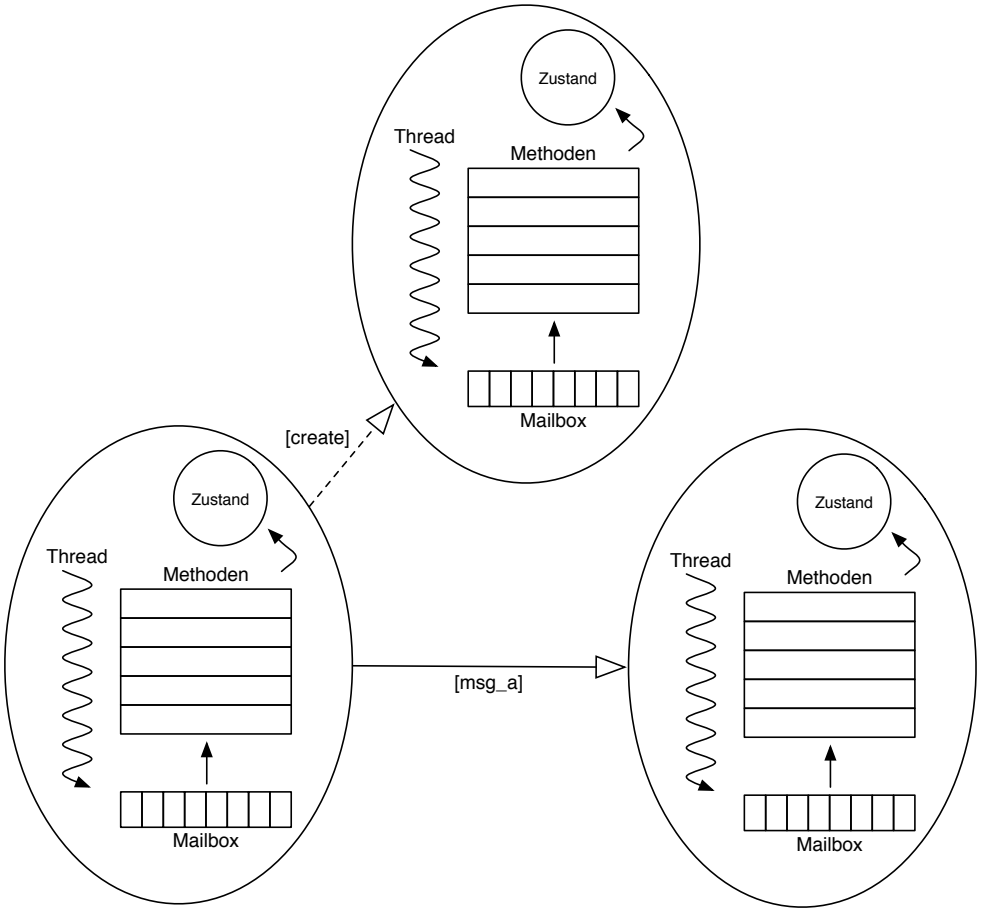


Abbildung: Komponenten des Aktor-Modells

Aktoren - Beispiel in *Scala* (1)

Aktoren in *Scala*:

- Erzeugung eines Aktors:
  - Methode `actor()` des `Actor`-Objekts
  - Eigene Klasse, welche von *Actor*-Klasse erbt
- Senden von Nachrichten an andere Aktoren:
  - `!(msg: Any): Unit` - asynchron
  - `!?(msg: Any): Any` - synchron
  - `!!(msg: Any): Future[Any]` - asynchron(*Futures*)
- Empfang von Nachrichten:
  - `react()` - kein Rückgabewert, ein Thread
  - `receive()` - Rückgabewert, je ein Thread
  - `reactWithin(msec: Long)` - wie `react`, mit Timeout
  - `receiveWithin(msec: Long)` - wie `receive`, mit Timeout

Aktoren - Beispiel in *Scala* (2)

Beispiel

```
val fussyActor = actor {
  loop {
    receive {
      case "quit" => println("ByeBye"); exit
      case s: String => println("I got a String: " + s)
      case i: Int => println("I got an Int: " + i)
      case _ => println("I have no idea what I just got")
    }
  }
}
```

Aktoren - Beispiel in *Scala* (2)

Aufruf

```
fussyActor ! "hi there"
fussyActor ! 23
fussyActor ! 3.33
fussyActor ! "quit"
```

Ausgabe

```
I got a String: hi there
I got an Int: 23
I have no idea what I just got
ByeBye
```

Aktoren - Beispiel mit Pipeline Entwurfsmuster (1)

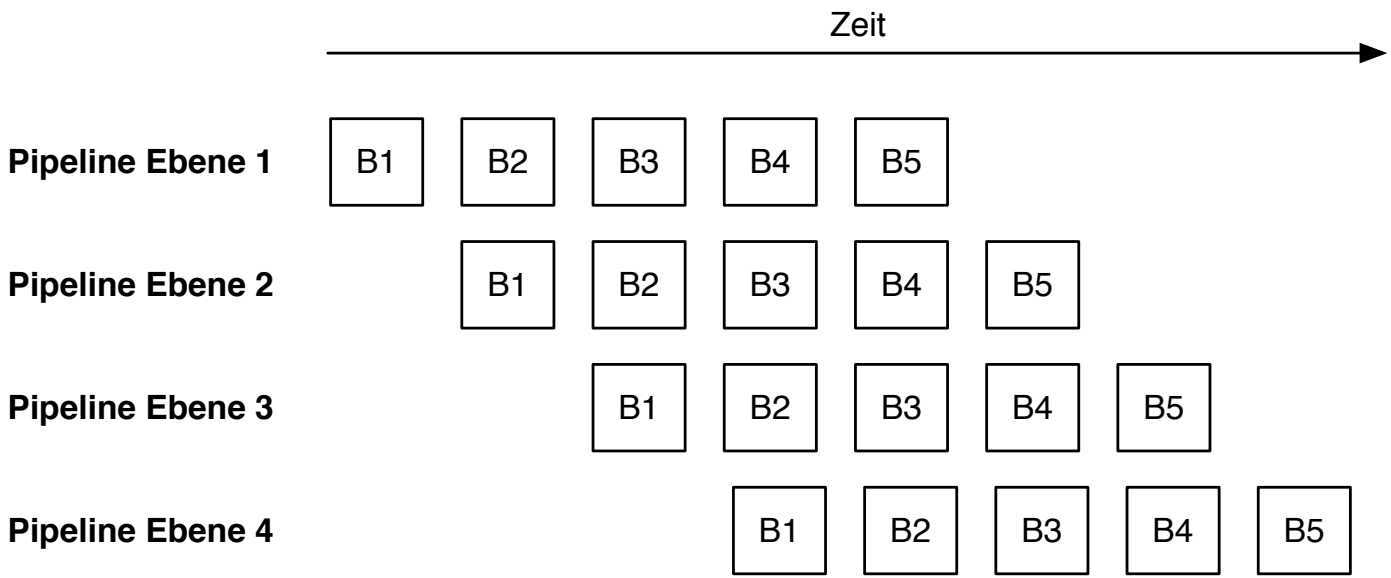


Abbildung: Pipeline

Aktoren - Beispiel mit Pipeline Entwurfsmuster (2)

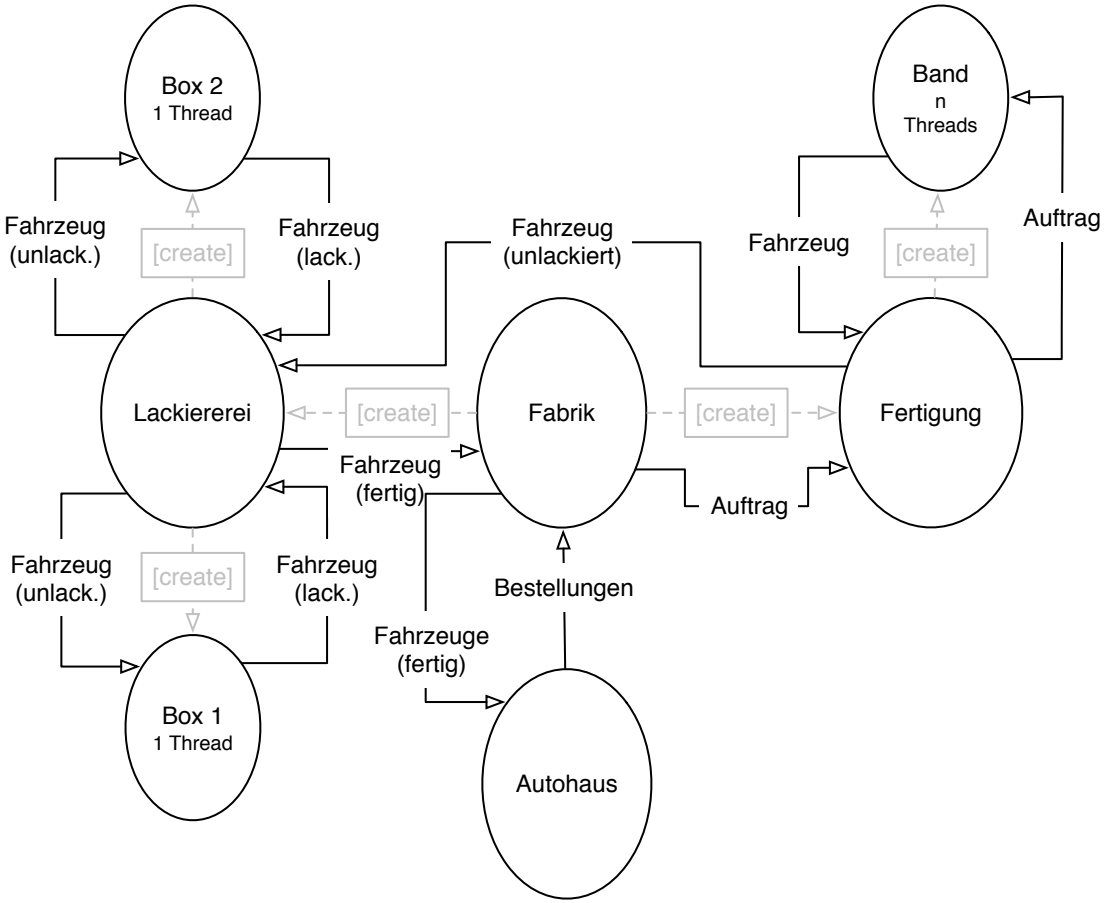


Abbildung: Beispiel: Fahrzeugfabrik

Aktoren - Perfekt für Architekturen ohne Cache-Kohärenz

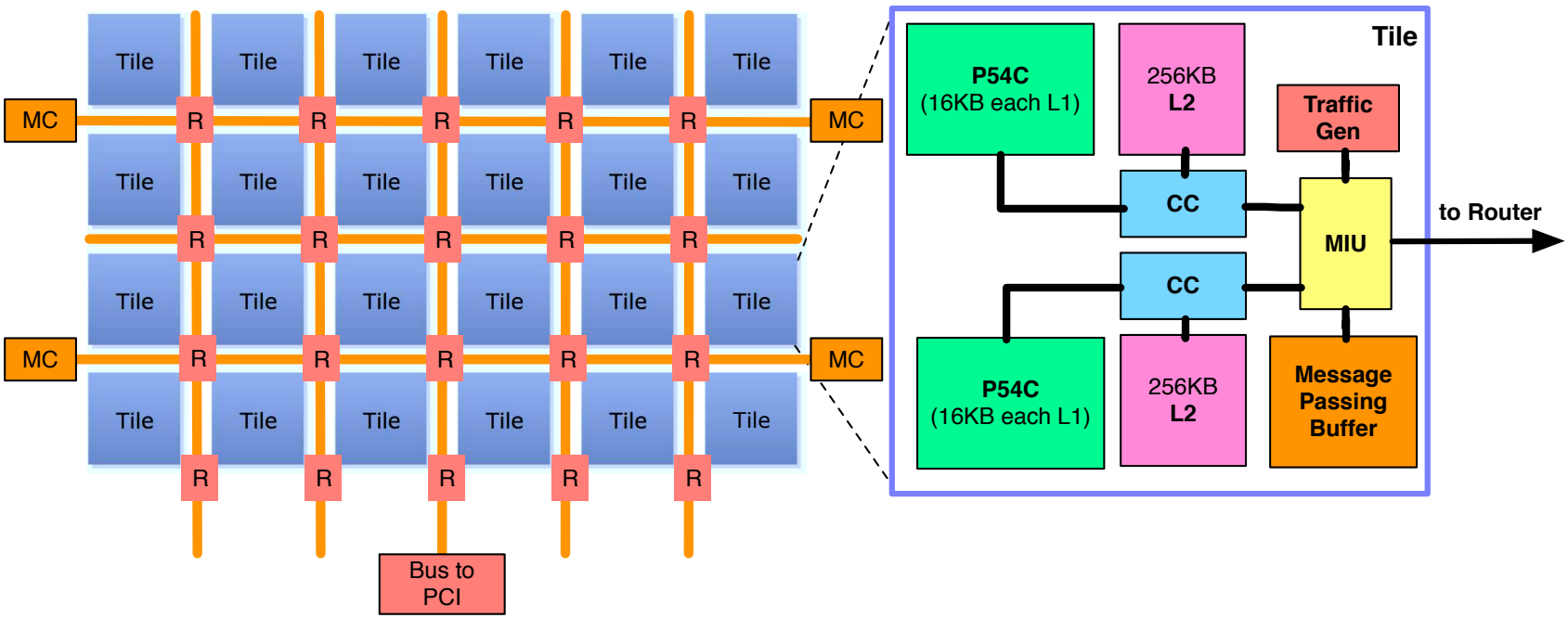


Abbildung: Intel *Single-chip Cloud Computer* [2]

[2] http://techresearch.intel.com/

Aktoren - Fazit

Vorteile

- Hohe Abstraktion von Parallelität
- Gut für skalierbare, reaktive Systeme
- Optimal für Plattformen mit verteiltem Speicher

Nachteile

- Nachrichtenübermittlung kann viel Overhead verursachen
- Manuelle Steuerung von Nebenläufigkeit schwierig

Autotuner

Erläuterung

Autotuner untersuchen automatisch den Optimierungs-Raum einer Applikation auf einer speziellen Hardware-Architektur. Dadurch kann die beste Kombination an Algorithmen, Implementierungen und Datenstrukturen für die Eingabedaten bestimmt werden.

Vorgehen:

1. Suche nach möglichen Implementierungen
2. Quellcode-Generierung
3. Vergleich der erstellten Binaries

Autotuner - Fazit

Vorteile

- Optimale Effizienz durch syntaktische und semantische Informationen
- Ermöglichen Portabilität

Nachteile

- Erfordern viel Rechenaufwand
- Komplex

Transaktionsspeicher (1)

Definition (Transaktion)

Eine Transaktion ist eine endliche Sequenz von Instruktionen, die von einem einzelnen Thread ausgeführt werden

Folgende Eigenschaften werden erwartet:

1. Atomarität
2. Konsistenz
3. Isolation

Transaktionsspeicher (2)

Hardware-Transaktionsspeicher

- Zugriff auf gemeinsame Speicherbereiche durch Cache-Änderungen
- Bei Datenkonflikt: Neustart der Transaktion
- Bei erfolgreicher Aktion: Übertragen der Cache-Daten

Software-Transaktionsspeicher

- Isolation (stark - schwach)
- Update-Verfahren (*eager* - *lazy*)

Transaktionsspeicher - Fazit (1)

Vorteile

- Keine Synchronisation nötig

Nachteile

- Hardware-Transaktionsspeicher: begrenzte Cache-Größe
- Software-Transaktionsspeicher: Performanz

Transaktionsspeicher - Fazit (2)

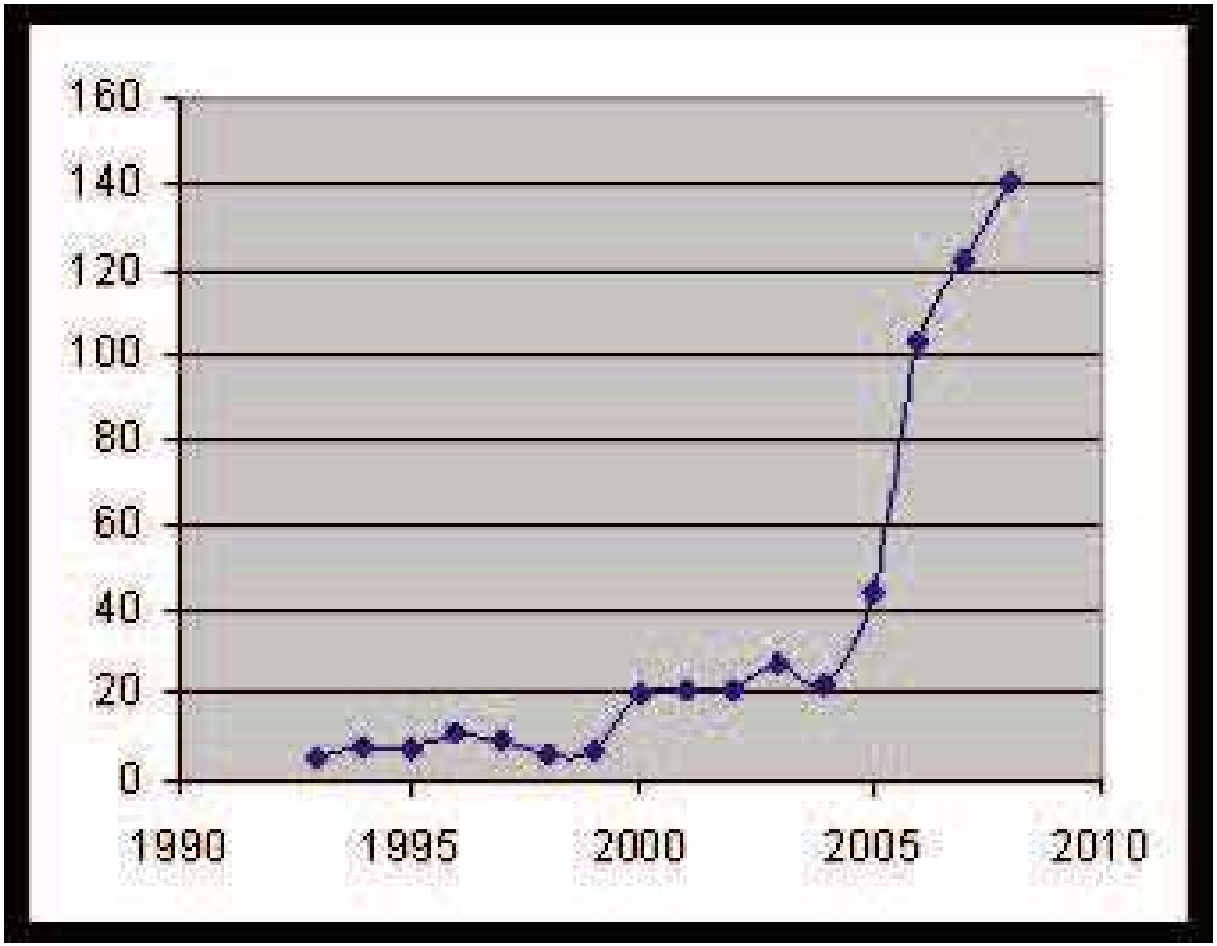


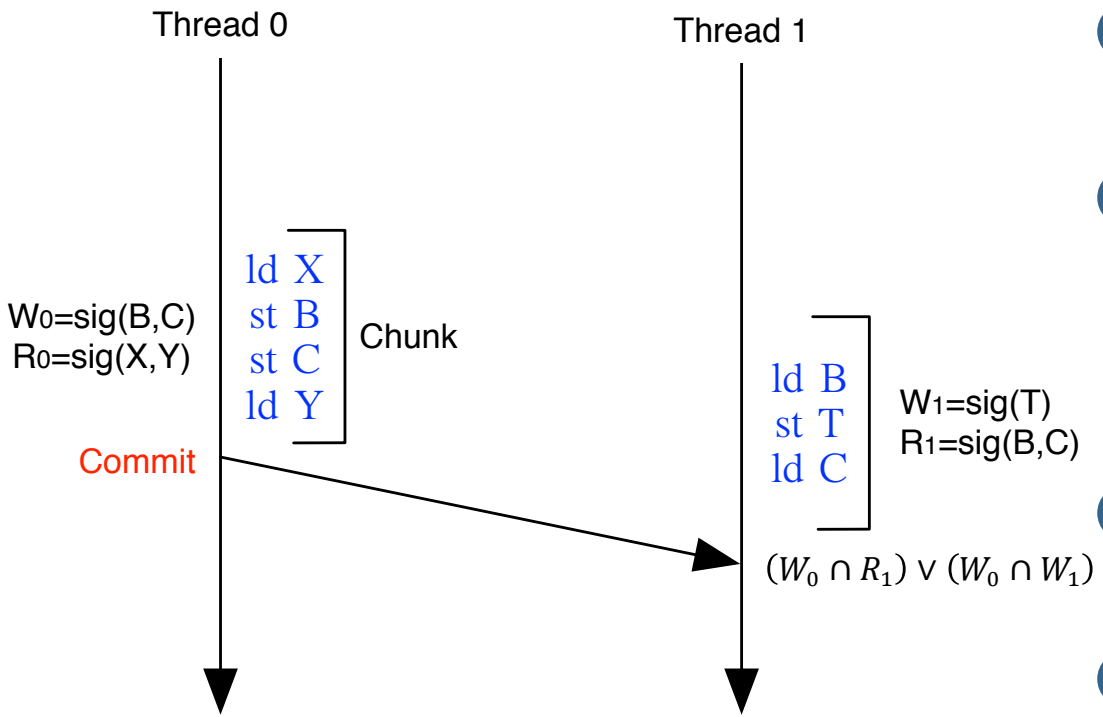
Abbildung: Jährliche Zitierungen auf Original-Arbeit[3]

[3] Transactional Memory Today - Maurice Herlihy

Bulk Multicore-Architektur - Vorstellung

- Wurde an der Universität Illinois entwickelt
- Ziel: Gute Programmierbarkeit und Skalierbarkeit
- Software als Serie atomarer Blöcke (*Chunks*)
- Verwendung von Hardware-Adress-Signaturen

Bulk Multicore-Architektur - Verarbeitung



1. Verarbeitung (Thread 0 $\rightarrow$ Thread 1)
2. Verarbeitung von Thread 0 fertig $\rightarrow$ Commit, *W*-Signatur an Thread 1
3. Vergleich der Signaturen
4. Konflikt $\rightarrow$ Neustart *Chunk* auf Thread 1

Abbildung: *Chunk*-Verarbeitung

Bulk Multicore-Architektur - Fazit

Vorteile

- Sequentielle Anwendungen werden “parallelisiert”
- Sequentielle Konsistenz gewährleistet

Nachteile

- Keine “objektiven” Praxiserfahrungen vorhanden
- Keine “wahre” Parallelisierung

Abschließendes Fazit (1)

- **Domänenspezifische Umgebungen**
  Hohe Abstraktion durch Domänen-Wissen
  Begrenztes Einsatzgebiet
- **Parallele Entwurfsmuster**
  Abstraktion durch bewährte Verfahren
  Erfahrung im Umgang notwendig
- **Parallele Skelette**
  Formale Schablonen sorgen für hohe Abstraktion
  Mangel an praktischen Implementierungen

Abschließendes Fazit (2)

- **Aktoren**
  Erstellung von reaktiven Systemen ohne Synchronisation
  Mangel an praktischen Implementierungen
- **Transaktionsspeicher**
  “Die” Lösung für Synchronisationsprobleme
  Noch nicht praxistauglich
- ***Bulk Multicore*-Architektur**
  Sequentielle Anwendungen werden automatisch parallelisiert
  Forschungsstatus

Ausblick

Vielen Dank für die Aufmerksamkeit!