

Verteilte und Parallele Programmierung

Mit Virtuellen Maschinen

PD Stefan Bosse

Universität Bremen - FB Mathematik und Informatik

Parallele Programmierung

Grundlagen der parallelen Programmierung

Unterscheidung zwischen Parallelisierung im Daten- und Kontrollfluss von Programmen

Prozessmodelle und Prozesskonstruktoren

Parallelisierungsklassen

Datenfluss

Der Datenfluss beschreibt den Fluss von Daten durch Verarbeitungs- und Speichereinheiten (Register).

Die Verarbeitungseinheiten sind Knoten eines gerichteten Graphens, die Kanten beschreiben den Datenfluss und bilden die Datenpfade. Die Verarbeitungseinheiten müssen aktiviert werden.

Kontrollfluss

Der Kontrollfluss beschreibt die temporale schrittweise Verarbeitung von Daten im Datenpfad durch zustandsbasierte selektive Aktivierung von Verarbeitungseinheiten.

Der Kontrollfluss kann durch **Zustandsübergangsdigramme** beschrieben werden.

Parallelisierungsklassen

Programmfluss

Der Programmfluss setzt sich kombiniert aus Daten- und Kontrollfluss zusammen.

Parallelisierungsklassen

- Programmanweisungen werden Zuständen $S_1, S_2, ..$ zugeordnet.
- Einfache Anweisungen (Berechnungen) werden jeweils einem Zustand, komplexe Anweisungen i.A. mehreren Unterzuständen zugeordnet.

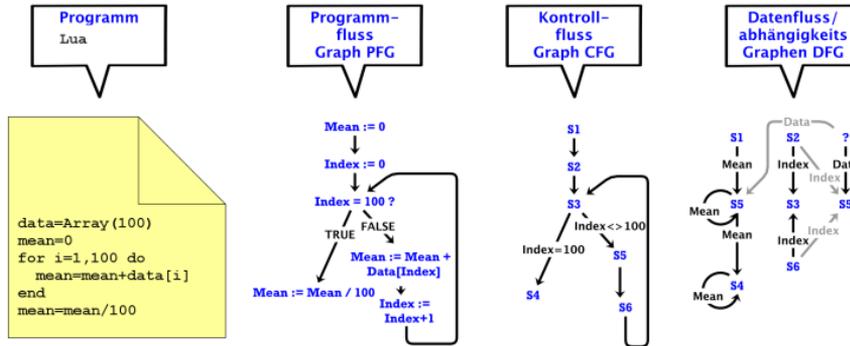


Abb. 1. Programm-, Kontroll-, und Datenflussgraphen

Parallelisierungsklassen

Datenparallelität

In vielen Programmen werden dieselben Operationen auf unterschiedliche Elemente einer Datenstruktur angewendet. Im einfachsten Fall sind dies die Elemente eines Feldes.

- Wenn die angewendeten Operationen unabhängig voneinander sind, kann diese verfügbare Parallelität dadurch ausgenutzt werden, um die zu manipulierenden Elemente der Datenstruktur auf verschiedene Prozessoren zu verteilen, so dass jeder Prozessor die Operation auf den ihm zugeordneten Elementen ausführt.
- Parallelität im Datenpfad → Feine Granularität

- Bei der Datenparallelität wird unterschieden zwischen:
 - Parallele Ausführung der gleichen Instruktion auf verschiedenen Daten (Vektorparallelität) → Vektoranweisung
 - Ausführung verschiedener Instruktionen die nur Daten verarbeiten (reine Datenanweisungen, keine Kontrollpfadverzweigungen)

Parallelisierungsklassen

- Zur Ausnutzung der Datenparallelität wurden sequenzielle Programmiersprachen zu datenparallelen Programmiersprachen erweitert. Diese verwenden wie sequenzielle Programmiersprachen einen Kontrollfluß, der aber auch datenparallele Operationen ausführen kann.
 - Z.B. μ RTL: Bindung von mehreren Datenpfadanweisungen durch Kommasyntax: $X \leftarrow \epsilon, Y \leftarrow \epsilon, .., Z \leftarrow \epsilon;$
- Häufig werden Vektoranweisungen deklarativ und nicht prozedural imperativ beschrieben.

Parallelisierungsklassen

- Beispiel für eine deklarative Vektoranweisung und die dazugehörige imperative Anweisungssequenz in Schleifenform (Fortran 90):

```
a(1 : n) = b(0 : n - 1) + c(1 : n) ⇔  
for (i=1:n)  
    a(i) = b(i-1) + c(i)  
end
```

Parallelisierungsklassen

- **Datenabhängigkeiten** können zwischen der linken und rechten Seite einer Datenzuweisung bestehen, so dass
 - zuerst auf alle auf der rechten Seite auftretenden Felder zugegriffen wird und die auf der rechten Seite spezifizierten Berechnungen durchgeführt werden,
 - bevor die Zuweisung an das Feld auf der linken Seite der Vektoranweisung erfolgt!
- Daher ist folgende Vektoranweisung nicht in die folgende Schleife sequenziell transformierbar:

```
a(1 : n) = a(0 : n - 1) + a(2 : n + 1) ≠  
for (i=1:n)  
  a(i) = a(i-1) + a(i+1)  
end
```

Parallelisierungsklassen

Datenparallelität benötigt i.A. keine weitere Synchronisation

Aber Datenabhängigkeiten können zu einer impliziten oder expliziten Synchronisation führen (z.B. mit Queues)



Parallelisierungsklassen

Instruktionsparallelität

- Parallelität im Kontrollpfad (Zustandsautomat) auf Instruktions- und Prozessebene → Grobe Granularität je nach Anzahl der Instruktionen pro Prozess (Task)
- Gebundene Instruktionsblöcke sind Parallelprozesse (*Par* Prozesskonstruktor)
- Bekannt als Multithreading als Programmier- und Ausführungsmodell (z.B. *pthread*)
- Instruktionsparallelität benötigt i.A. Synchronisation zwischen den einzelnen Prozessen

Parallelisierungsklassen

Möglichkeiten der Parallelisierung

1. Vertikale Parallelität auf Bitebene (jegliche funktionale Operation)
2. Horizontale Parallelität durch Pipelining (nur Datenströme)
3. Parallelität durch mehrere Funktionseinheiten:
 - Superskalare Prozessoren
 - Very Large Instruction Word (VLIW) Prozessoren
4. Funktionsparallelität (Evaluierung der Funktionsargumente und Rekursion)
5. Vertikale Parallelität auf Prozess- bzw. Threadebene (Kontrollpfadebene)

Fluss und Pfadgraphen

- Ein **Datenfluss** (Graph aus Operationen oder Variablen) beschreibt den Verlauf und die Verarbeitung von Daten durch Operationen → **Verhaltensbeschreibung**
- Ein Datenpfad beschreibt die Verbindung von Komponenten zur Implementierung des Datenflusses → **Strukturbeschreibung**
- Der Kontrollfluss beschreibt die zustandsbasierte Steuerung einer Berechnung
 - Bedingte (\leftrightarrow Datenfluss) Verzweigungen im Kontrollfluss
 - Schleifen im Kontrollfluss
- Der Kontrollpfad implementiert den Kontrollfluss → **Zustandsautomat**

Daten- und Kontrollpfade

Jeder generische Mikroprozessor und i.A. jedes anwendungsspezifische Digitallogiksystem lässt sich in die zwei funktionale Bereiche aufteilen:

1. Datenfluss → Datenpfade
2. Kontrollfluss → Kontrollpfad → Zustandsautomat

Datenpfad

- Führt Berechnungen durch
- Kann rein funktional (speicherlos, kombinatorisch) oder speicherbasiert mit Register-Transfer Architektur sein

Daten- und Kontrollpfade

Kontrollpfad

- Der Kontrollpfad ist immer zustandsbasiert (Speicher) und bestimmt die **Reihenfolge** von Operationen und Berechnungen

Daten- und Kontrollpfade

Parallelisierung

Der Programmfluss kann in Daten- und Kontrollfluss zerlegt werden, die jeweils im Daten- und Kontrollpfad eines Datenverarbeitungssystems verarbeitet werden.

Kontrollpfad

Parallelität auf Prozessebene (Multithreading) mit Interprozesskommunikation → Mittlere bis geringe Beschleunigung, mittlerer Overhead

Datenpfad

Parallelität auf Dateninstruktionsebene ohne explizite Kommunikation → Hohe Beschleunigung, geringer Overhead

Daten- und Kontrollpfade

Programm

- Ein (prozedurales/imperatives) Programm besteht aus mindestens einem Daten- und Kontrollpfad (oder Fluss)
- Ein Programm ist aus einer Vielzahl von Komponenten zusammengesetzt → **Komposition**
- Komposition von Datenfluss und Datenpfad:
 - **Funktionale Komposition**
 - (Sequenzielle Komposition)
 - **Parallele Komposition**

Daten- und Kontrollpfade

- Komposition von Daten- und Kontrollfluss/pfad:
 - **Sequenzielle Komposition**
 - **Parallele Komposition**

Funktionale Programmierung

Funktionale Programmierung beschreibt grundlegend den Datenpfad. Es gibt keinen Zustand und keinen expliziten Kontrollfluss.

Funktionale Komposition

Eine Berechnung setzt sich aus der verschachtelten und verketteten Applikation von Funktionen zusammen, die jeweils aus elementaren Ausdrücken $E = \{\varepsilon_1, \varepsilon_2, \dots\}$ bestehen

→ **Funktionale Komposition** $F(\mathbf{X}) = f_1(f_2(f_3(\dots(\mathbf{X}))))$

- Jede Funktion f_i beschreibt einen Teil der Berechnung.

Funktionale Komposition

- Ausdrücke bestehen aus:

1. Konstante und variable Werte
1, 2.0, 'c', "hello", x, y, z
2. Einfache arithmetische
Ausdrücke $\epsilon = x + y$
3. Zusammengesetzte Ausdrücke
(arithmetische Komposition)
 $\epsilon = (x + 1) * (y - 1) * z$
4. Relationale Ausdrücke $x < 0$
5. Boolesche Ausdrücke a and b
or c
6. Bedingte Ausdrücke liefern
Werte if $x < 0$ then $x + 1$
else $x - 1$
7. Funktionsapplikation $f(x, y, z)$

Funktionale Komposition

- Entspricht dem mathematischen Modell was auf Funktionen und Ausdrücken gründet mit den Methoden
 - **Definition** $f(x) = \lambda.x \rightarrow \varepsilon(x)$
 - **Applikation** $f(\varepsilon)$, bei mehreren Funktionsargumenten $f(\varepsilon_1, \varepsilon_2, \dots)$
 - **Komposition** $f \circ g \circ h \circ \dots = f(g(h(\dots(x))))$
 - **Rekursion** $f(x) = \lambda.x \rightarrow \varepsilon(f, x)$
 - und **Substitution** $a = \varepsilon$
- Zeitliches Modell: unbestimmt ($t \mapsto 0$), *Auswertereihenfolge nicht festgelegt*

Funktionale Komposition

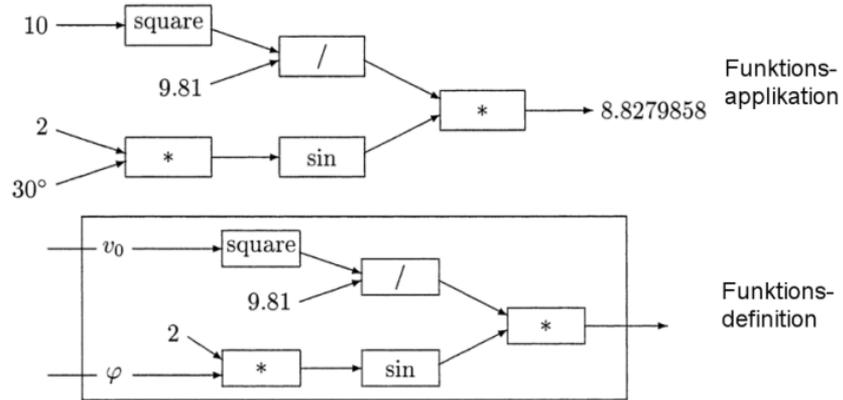


Abb. 2. Beispiel für Funktionsdefinition und Applikation

Funktionale Komposition von Systemen

Parallelisierung durch Modellierung unterschiedlich detaillierter Datenflussdiagramme, die untereinander einen hierarchischen Zusammenhang aufweisen.

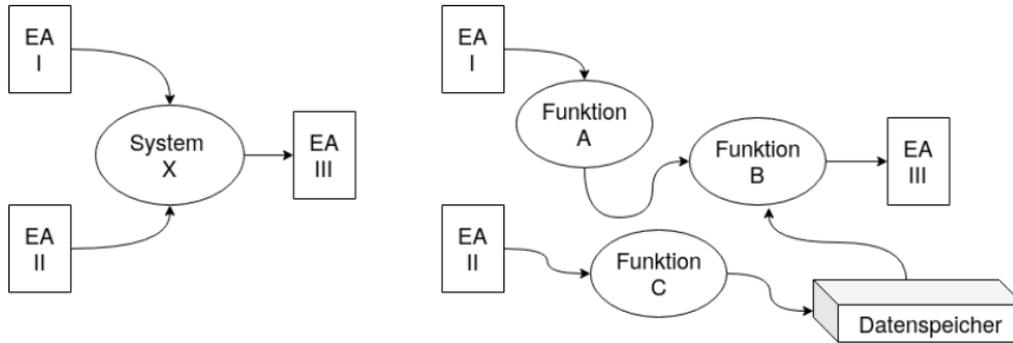


Abb. 3. (Links) Datenflüsse von Ein- und Ausgabegeräten EA eines Systems (Rechts) Komposition des Systems durch Funktionen mit inneren Datenflüssen (teils parallelisierbar)

Sequenzielle Komposition

Eine imperative Berechnung setzt sich aus einer Sequenz $\mathbf{I}=(i_1, i_2, ..)$ von elementaren Anweisungen $\mathbf{A}=\{a_1, a_2, ..\}$ zusammen \rightarrow

Sequenzielle Komposition $i_1 ; i_2 ; \dots$

- Die Anweisungen werden sequentiell in *exakt* der angegebenen Reihenfolge ausgeführt

- Die Menge der Anweisungen **A** lässt sich in folgende Klassen unterteilen:
 1. Arithmetische, relationale, und boolesche **Ausdrücke** (in Verbindung mit 2./3.) $x+1$, $a*(b-c)*3$, $x<(a+b)$, ...
 2. **Datenanweisungen**: Wertzuweisungen an Variablen $x := a+b$
 3. **Kontrollanweisungen** wie bedingte Verzweigungen und Sprünge (Schleifen)

```
if a < b then x := 0 end  
while x < 100 do x:=x+1 end
```

Sequenzielle Komposition

- Man fasst die Sequenz **I** als ein imperatives **Programm X** == Ausführungsvorschrift zusammen.
- Die **Ausführung** des Programms (statisch) bezeichnet man als **Prozess** (dynamisch)
- Ein Prozess befindet sich aktuell immer in einem **Zustand** σ einer endlichen Menge von Zuständen $\Sigma = \{\sigma_1, \sigma_2, \dots\}$.
- Der gesamte Zustandsraum Σ des Prozesses setzt sich aus dem
 - **Kontrollzustand** $s \in \mathbf{S} = \{s_1, s_2, \dots\}$, und dem
 - **Datenzustand** mit der Menge aller Speicherzustände (Daten) $d \in \mathbf{D} = \{d_1, d_2, \dots\}$ zusammen.

Sequenzielle Komposition

- Der Fortschritt eines sequenziellen Programms bedeutet ein Änderung des Kontroll- und Datenzustandes → **Zustandsübergänge**
- Die Zustandsübergänge können durch ein Zustandsübergangsdiagramm dargestellt werden.

Das Prozessmodell

Es werden zwei Prozesstypen unterschieden:

Sequenzieller Prozess

Ein einziger Programm- und Kontrollfluss mit strikter sequenzieller Ausführung von Anweisungen

Paralleler Prozess

Es gibt zwei oder mehr parallel ausgeführte Programm- und Kontrollflüsse (Ausführung kann zeitlich überlappend sein, muss aber nicht → Scheduling)

Sequenzieller Prozess

Ausführungszustände

- Ein Prozess P besitzt einen "makroskopischen" **Metaausführungszustand** ρ_S . Die Basismenge der Ausführungszustände **PS** ist ("Milestones"):
 - Start/Bereit aber noch nicht rechnend (START)
 - Rechnend (RUN)
 - Terminiert und nicht mehr rechnend (END)
- Dazu kann es eine erweiterte Menge **PS*** an Metaausführungszuständen geben:
 - Auf ein Ereignis wartend (AWAIT)
 - Blockiert (BLOCKED)
 - Rechenbereit aber nicht rechnend (nach Ereignis) (READY)

$$PS = \{\text{START}, \text{RUN}, \text{END}\}$$

$$PS^* = \{\text{AWAIT}, \text{BLOCKED}, \text{READY}\}$$

$$ps(P) \in PS \cup PS^*$$

- Befindet sich der Prozess im Ausführungszustand RUN (also ausführend), dann werden strikt sequenziell die Instruktionen des Programms $\mathbf{I} = \{i_1, i_2, \dots, i_n\}$ ausgeführt.
- Dabei ist jede Instruktion eine Anweisung aus der endlichen Anweisungsmenge $\mathbf{A} = \{a_1, \dots\}$ der Maschine (z.B. Bytecode Anweisung $\text{add}(x, y)$).
- Ein Wechsel des Ausführungszustandes führt nicht zu einer Änderung der Instruktionsreihenfolge.

Sequenzieller Prozess: Zeitliches Modell

- Ein Prozess führt die Anweisungen der Reihe nach aus. Das symbolische Semikolon teilt den Anweisungen eigene Ausführungsschritte (Zeitpunkte t_i) zu.

$$i_1 : t_1 \mapsto i_2 : t_2 \mapsto i_3 : t_3 \mapsto \dots \mapsto i_n : t_n \text{ mit}$$
$$t_1 < t_2 < t_3 < \dots < t_n$$
$$T = \sum_i t_i - t_{i-1}$$

- Die gesamte Ausführungszeit $T(p)$ eines sequenziellen Prozesses p ist die Summe der Ausführungszeiten der Elementarprozesse

Sequenzieller Prozess

Prozessblockierung

- Das Konzept der Prozessblockierung ist elementar für kommunizierende Prozesse
- Kommunikation: Lesen und Schreiben von Daten, Ein- und Ausgabe, Netzwerknachrichten, usw.
- Befindet sich ein Prozess im Ausführungszustand BLOCKED wartet dieser auf ein Ereignis:
 - Daten sind verfügbar
 - Zeitüberschreitung
 - Geteilte Ressource ist frei
 - Synchronisation mit anderen Prozessen

Sequenzieller Prozess

- Bei einem blockierten Prozess schreitet der Kontrollfluss nicht weiter voran (keine Terminierung der aktuellen Operation)
- Die Ausführung einer Operation (Instruktion) kann verzögert werden bis das zu erwartende Ereignis eintritt

Das Ausführungsmodell von Lua unterstützt die Blockierung des Programmflusses

Koroutinen

- In Koroutinen (Fiber) z.B. mittels der *yield* Anweisung
- Der Programmfluss wird über einen Scheduler an eine andere Koroutine übergeben
- Schließlich wird die Programmausführung der ursprünglichen Koroutine nach der *yield* Anweisung wieder fortgesetzt (durch *resume* Anweisung)

Sequenzieller Prozess

- Ebenso auch bei Ein- und Ausgabeoperationen (IO: *read*, *write*)
- In Threads ebenfalls via *yield* und IO Operationen
 - Aber hier findet das Scheduling außerhalb der VM statt (multiple VM Instanzen)

Symmetrische Koroutinen

Das Scheduling der Koroutinen erfolgt semiautomatisch und es gibt nur die *yield* Operation (Reaktivierung von Koroutinen implizit durch Scheduler).

Asymmetrische Koroutinen

Die Suspendierung und Reaktivierung von Koroutinen erfolgt explizit (*yield* und *resume* Operationen).

Sequenzieller Prozess

Tatsächlich gibt es in Lua auf programmatischer Ebene aber nur **einen einzigen Kontrollfluss** und Pfad (eine VM Instanz)!

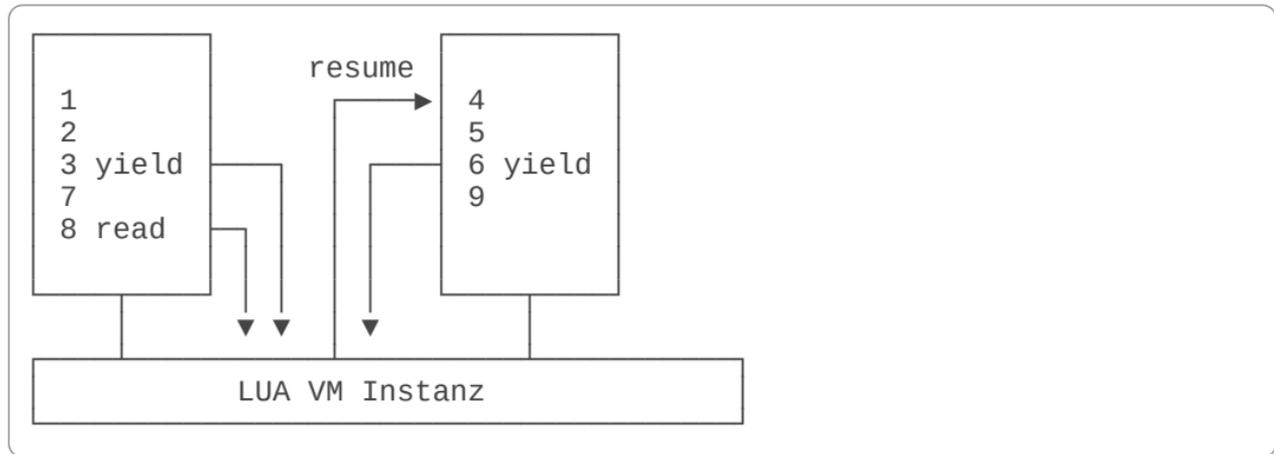


Abb. 4. Zwei Koroutinen im Wechsel; eine VM Instanz

Sequenzieller Prozess

```
co1 = coroutine.create(function ()
    for i=1,10 do
        print("co1", i)
        coroutine.yield()
    end
end)
co2 = coroutine.create(function ()
    for i=1,10 do
        print("co2", i)
        coroutine.yield()
    end
end)
coroutine.resume(co1)
coroutine.resume(co2)
...
```

Parallel LuaJit Virtual Machine (LVM)

Parallele Prozesse

Ausgangspunkt sind sequenzielle Prozesse

Ein sequenzieller Prozess P_s besteht aus einer Sequenz von Anweisungen $\mathbf{I}=(i_1 ; i_2 ; ..)$ (Berechnungen), die schrittweise und nacheinander ausgeführt werden: $P_s=P(i_1 ; i_2 ; .. ; i_n) \Rightarrow p_1 \mapsto p_2 \mapsto .. \mapsto p_n$

Parallele Komposition

Datenverarbeitungssystem als paralleler Prozess kann aus einer Vielzahl nebenläufig ausgeführter Prozesse $P=\{P_1, P_2, P_3, \dots\}$ bestehen

Parallele Komposition $P=P_1 \parallel P_2 \parallel P_3 \parallel \dots$

Paralleler Prozess

Prozesskonstruktor

$$P = (P_1 \parallel P_2 \parallel \dots \parallel P_n) \Leftrightarrow$$

PAR

i₁

i₂

..

i_n

Lua

```
Par({
  function (pid) .. end,
  function (pid) .. end,..
},{
  -- shared environment
  v = ε, ..
})
```

Paralleler Prozess

Ausführungsreihenfolge

- Achtung! Wenn keine Kommunikation (Synchronisation) zwischen den Prozessen stattfindet ist die Ausführung und das Endergebnis (der Berechnung) gleich der sequenziellen Ausführung der Prozesse:

$$P_1 \parallel P_2 \parallel \dots \parallel P_n \Rightarrow P_1; P_2; \dots; P_n$$

$$P_1 \parallel P_2 \Rightarrow P_1; P_2 \Rightarrow P_2; P_1!$$

Paralleler Prozess: Zeitliches Modell

$$P = P_1 \parallel P_2 \parallel \dots \parallel P_n$$

$$P_1 = (i_{1,1} : t_{1,1}; i_{1,2} : t_{1,2}; \dots) \dots P_n = (i_{n,1} : t_{n,1}; i_{n,2} : t_{n,1}; \dots), \text{ mit}$$

$$t_{i,1} < t_{i,2} < \dots < t_{i,m} \forall \text{ Prozess mit } m \text{ Instr. und}$$

$$t(P) = [t_1, t_2] = [t_{1,1}, t_{1,q}] \cup [t_{2,1}, t_{2,r}] \cup \dots \cup [t_{n,1}, t_{n,s}]$$

$$t_1 = \min\{t_{1,1}, t_{2,1}, \dots, t_{n,1}\}$$

$$t_2 = \max\{t_{1,q}, t_{2,r}, \dots, t_{n,s}\} \quad T = t_2 - t_1$$

Prozessflussdiagramme

- Der Prozessfluss in einem parallelen System kann durch Übergangs- und Terminierungsdiagramme dargestellt werden. (wie bereits im seq. Fall gezeigt wurde).
- Eingehende Kanten (Pfeile) des Graphen beschreiben den Start eines Prozesses, ausgehende Kanten die Terminierung.

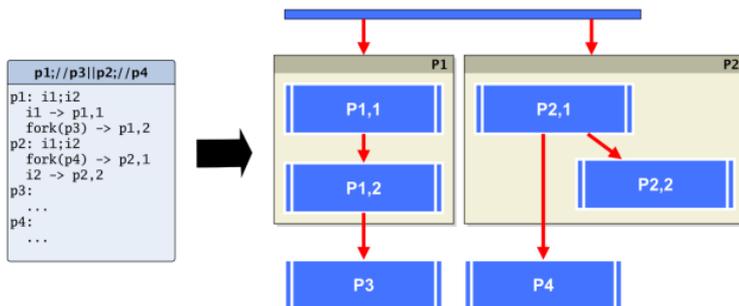


Abb. 5. Beispiel eines Prozessflussdiagramms für vier Prozesse. P_1 startet P_3 , und P_2 spaltet P_4 ab. (mit $P_{,i} = \text{Instruktion}(P, i)$)

Paralleler Prozess

Prozesserzeugung und Synchronisation

- Ein neuer paralleler Prozess kann durch den entsprechenden Prozesskonstruktor erzeugt werden
- Geschieht dies innerhalb eines Prozesses P_i so wird dessen Prozessausführung blockiert *bis* alle Teilprozesse des parallelen Prozesses terminiert sind!

Forking

- Ein neuer Prozess P_j kann mittels Forking erzeugt werden (Aufspaltung des Kontrollflusses, so auch von einem anderen Prozess P_i) ohne dass ein ausführender Prozess auf die Terminierung des Parallelprozesses warten muss:

$$P_1; // P_2; P_3 = P_1; P_3 \parallel P_2$$
$$// P_1; // P_2; P_3 = P_1 \parallel P_2 \parallel P_3$$

Lua

```
Fork({  
  function (pid) .. end,  
  function (pid) .. end, ..  
}, {  
  -- shared environment  
})
```

Parallele Prozesse in Lua

Parallel LuaJit Virtual Machine (LVM)

Zusammenfassung

Parallele Prozesse werden aus nebenläufig oder verwoben ausgeführten sequenziellen Prozessen gebildet

Prozesse besitzen drei wesentliche Ausführungszustände: Bereit, Rechnend, Terminiert

Prozesssynchronisation kann durch Prozessflussgraphen (PFG) dargestellt werden

Ein nachfolgender Prozess (Knoten im PFG) wird erst gestartet wenn der vorherige terminiert

Prozessblockierung ist wesentliche Eigenschaft von synchronisierenden Prozessen