Programmierparadigmen

Kapitel 4b
Parallele Programmierung in Erlang

(Diese Folien beruhen auf einem Foliensatz von Prof. Dr. Kai Uwe Sattler)

Überblick

- Grundlagen
- Parallele Programmierung in Erlang
  - Prozesse
  - Datenparallelität
  - Taskparallelität
- Parallele Programmierung in C++
- Parallele Programmierung in Java

Unterstützung paralleler Programmierung in Erlang

- Leichtgewichtige Prozesse und Message Passing
- SMP-Support
- Ziele für effiziente Parallelisierung:
  - Problem in viele Prozesse zerlegen (aber nicht zuviele …)
  - Seiteneffekte vermeiden (würde Synchronisation erfordern)
  - Sequentiellen Flaschenhals vermeiden :
    - Zugriff auf gemeinsame Ressourcen: IO (Input/Output), Registrierung von Prozessen, ...

Prozesse in Erlang

- Erlang VM = Betriebssystemprozess
- Erlang-Prozess = Thread innerhalb der Erlang VM
  - Kein Zugriff auf gemeinsame Daten, daher „Prozess“ genannt
- Jede Erlang-Funktion kann Prozess bilden
- Funktion **`spawn`** erzeugt einen Prozess, der die Funktion Fun ausführt
  - `Pid = spawn(fun Fun/0)`
- Resultat = Prozessidentifikation Pid, mittels der man dem Prozess Nachrichten schicken kann
- Über **`self()`** kann man die eigene Pid ermitteln
- Übergabe von Argumenten an den Prozess bei der Erzeugung
  - `Pid = spawn(fun() -> any_func(Arg1, Arg2, ...) end)`

Prozesse in Erlang: Beispiele

```
1> spawn(fun() -> io:format(["Hallo Erlang!"]) end).
Hallo Erlang!<0.133.0>
```

- Es wird ein Thread gestartet, der eine anonyme Funktion ausführt, welche die Funktion `io:format(…)` ausführt

```
-module(ch4_1).

-export([start/0, say_hello/1]).

say_hello(Msg) ->
    io:format("~n~p", [Msg]).

start() ->
    spawn(ch4_1, say_hello, ["Hallo Erlang!"]).
```

- Die vom Modul exportierte Funktion `start`, startet einen Thread, der die Funktion `say_hello` mit Parameterliste [„Hallo Erlang“] ausführt

SMP mit Erlang

```
Erlang/OTP 23 [erts-11.0] [source] [64-bit] [smp:4:4]

Eshell V11.0         (abort with ^G)
1> erlang:system_info(schedulers).
4
```

- Die Erlang-VM läuft in diesem Beispiel mit 4 OS-Threads, die in diesem Fall auf 2 CPU-Kernen mit aktiviertem Hyperthreading laufen
- Die Zahl der Threads kann mit `-smp [disable | enable | auto]` beeinflusst werden
- Der optionale Parameter +S `[Anzahl]` legt die Anzahl der Scheduler fest:
  - Jeder Scheduler läuft in einem eigenen OS-Thread der Erlang-VM
  - Jeder Scheduler bearbeitet „Erlang Threads“ (≠ OS-Threads), die in einer oder mehreren „Run Queues“ verwaltet werden
  - Es sollten nicht mehr Scheduler gestartet werden als es CPU-Cores gibt

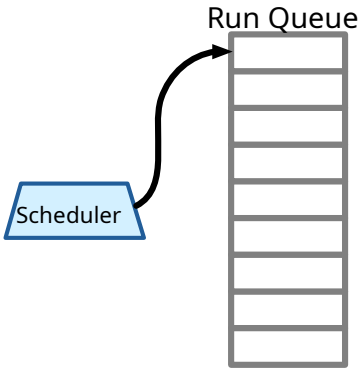
Erlang VMs, Nodes, Threads (1)

- Erlang-Programme laufen in sogenannten „Erlang Virtual Machines“, die in jeweils einem eigenen Betriebssystem-Prozess ablaufen
- Man kann auf einem Rechner mehrere Erlang-VMs starten, jede dieser VMs wird auch als Erlang-Node bezeichnet
- „Thread“ ist im Kontext von Erlang in zwei Bedeutungen zu verstehen:
  - Betriebssystem-Threads („OS-Threads“): beherbergen Erlang-VMs
  - „Erlang-Threads“: sind Erlang-eigene Datenstrukturen, die auszuführende Aufgaben beschreiben und die auch „Erlang-Prozesse“ genannt werden, da sie keinen Shared Memory unterstützen
- Erlang-Threads werden in einer oder mehrerer Warteschlangen („Run-Queues“) verwaltet
- Einer oder mehrere Scheduler entnehmen jeweils den/die nächsten zu bearbeitenden Erlang-Threads aus einer Run-Queue und fügen diese ggf. wieder nach Bearbeitung einiger Schritte in die Queue ein

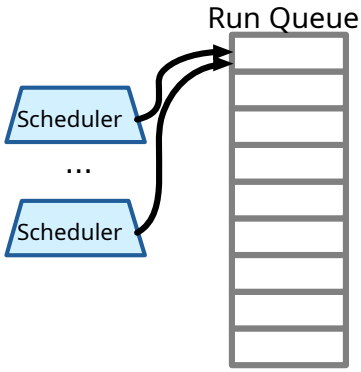
Erlang VMs, Nodes, Threads (2)

- Erlang unterstützt pre-emptives („verdrängendes“) Multitasking:
  - Wenn ein Erlang-Prozess auf eine Nachricht, I/O-Operation o.ä. wartet oder aus anderen Gründen vorerst nicht weiter berechnet werden kann/will, dann wird er unterbrochen und wieder in die Run-Queue eingefügt (→ „cooperative multitasking“)
  - Arbeitet ein Erlang-Prozess nach einer bestimmten Anzahl sogenannter „Reductions“ (~ Funktionsaufrufe, Default-Wert ist 2000) immer noch, dann wird er vom Scheduler unterbrochen und wieder in die Run-Queue eingefügt (→ „preemptive multitasking“)
- Erlang-Threads können unterschiedliche Prioritäten zugeordnet werden:
  - `PID = spawn(fun() -> process_flag(priority, high), … end).`
  - Unterstützte Prioritäten: `low | normal | high | max`
  - Der/die Scheduler berücksichtigen diese Prioritäten

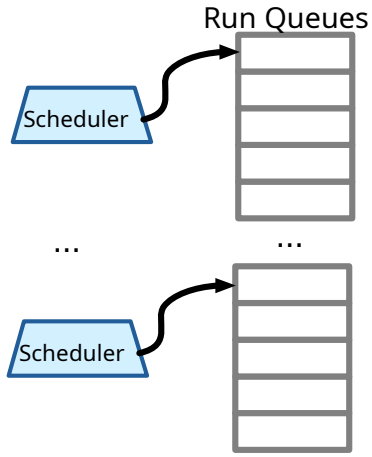
Schedulers in Erlang



Erlang VM
Single Scheduler
Singe Queue
(before R11B)



Erlang VM
Multiple Schedulers
Single Queue
(R11B/R12B)



Erlang VM
Multiple Schedulers
Multiple Queues
(since R13B)

- Bei Multiple Queues gibt es eine zusätzliche „Migration Logic“ für die Verschiebung von Erlang-Prozessen zwischen Queues

Message Passing in Erlang: Senden einer Nachricht

- Sämtliche Interaktion (Koordination, Datenaustausch) zwischen Erlang-Prozessen findet über Nachrichtenaustausch statt (es gibt keine gemeinsamen Variablen/Speicherbereiche zwischen Prozessen)
- Mittels Pid ! Message kann dem Prozess mit der Prozessidentifikation *Pid* eine Nachricht gesendet werden
- Dieser Befehl kann an beliebigen Stellen eines Ausführungsteils einer Erlang-Funktion verwendet werden
- Senden ist eine asynchrone Operation, dass heißt der sendende Prozess kann seine Berechnung sofort fortsetzen
- Damit der Prozess mit Identifikation *Pid* die Nachricht erhält, muss er eine Empfangsoperation ausführen

Message Passing in Erlang: Empfang von Nachrichten

```
1  receive
2      Pattern1 [when Guard1] -> Expressions1;
3      Pattern2 [when Guard2] -> Expressions2;
4      ...
5  end
```

- Trifft eine Nachricht ein, wird versucht, diese mit einem Pattern und ggf. vorhandenen Guard zu „matchen“
- Das erste zutreffende Pattern (inkl. Guard) bestimmt, welcher Ausdruck ausgewertet wird
- Trifft kein Pattern (inkl. Guard) zu, wird die Nachricht für mögliche spätere Verwendung aufgehoben und der Prozess wartet auf die nächste Nachricht (→ „selective receive“)

Ein einfacher Echo-Server

```
1  -module(ch4_2).
2  -export([run/0]).
3
4  run() -> Pid2 = spawn(fun loop/0),
5      Pid2 ! {self(), hello},
6      receive
7          {Pid2, Msg} -> io:format("P1 ~w~n",[Msg])
8      end,
9      Pid2 ! stop.
10
11 loop() ->
12     receive
13         {From, Msg} -> From ! {self(), Msg}, loop();
14         stop -> true
15     end.
```


Echo-Server: Erklärungen

- Die Funktion loop() realisiert einen (nur begrenzt nützlichen) Echo-Dienst, der jede emfangene Nachricht unverändert an den Absender zurück schickt, bis er nach Empfang des Atoms stop endet
- Die Funktion run()
  - startet den Echoserver (Zeile 4),
  - schickt ihm als nächstes eine Nachricht (Zeile 5),
  - wartet auf eine Antwort (Zeile 6),
  - gibt diese aus (Zeile 7) und
  - schickt dann das Atom stop an den Echo-Server
- Der rekursive Aufruf in der Funktion loop() erfolgt endrekursiv, daher wird kein wachsender Aufrufstapel angelegt (Hinweis: grundsätzlich zu beachten, da sonst der Speicherbedarf stetig wächst)

Ansätze zur Parallelisierung

- Beispiel: Berechnung der Fibonacci-Funktion für eine Liste zufällig generierter Eingabewerte und Rückgabe der Ergebnisse in einer Liste
- Sequentielle Lösung über list:map/2

```
1  % Berechnung der Fibonacci-Zahl für F
2  fibo(0) -> 0;
3  fibo(1) -> 1;
4  fibo(F) when F > 1 -> fibo(F - 1) + fibo(F - 2).
5
6  % Liste von Num Fibonacci-Zahlen
7  run(Num) ->
8      Seq = lists:seq(1, Num),          % durch Zufallszahlen ersetzen:
9      Data = lists:map(fun(_) -> random:uniform(20) end, Seq),
10     lists:map(fun fibo/1, Data).  % berechnet Ergebnisliste
```


Parallele Funktionen höherer Ordnung: pmap

- Parallele Variante von lists:map(Fun, List)
  - Für jedes Listenelement wird ein Prozess erzeugt
  - Anschließend werden die Ergebnisse „eingesammelt“

```
1  pmap(F, L) ->
2      S = self(), % Berechnung der Fibonacci-Zahl für F
3      Pids = lists:map(fun(I) ->
4                  % Prozess erzeugen
5                  spawn(fun() -> do_fun(S, F, I) end)
6                  end, L),
7      gather(Pids).    % Ergebnisse einsammeln
```

Joe Armstrong. *Programming Erlang: Software for a Concurrent World.* www.erlang-dach.org/blog/

Hilfsfunktionen für pmap – Verarbeitung von F

- Eigentliche Verarbeitungsfunktion ausführen und Ergebnis senden:

```
1  do_fun(Parent, F, I) ->
2      % Parent ist der Elternprozess
3    Parent ! { self(), (catch F(I))}.
```

- Bei dem Aufruf der Funktion F sorgt catch für die die korrekte Behandlung potentiell auftretender Fehler in F
- Das Ergebnis wird zusammen mit der eigenen Pid (durch self() berechnet) an den Elternprozess gesendet

Hilfsfunktionen für pmap – Ergebnisse einsammeln

```
1  % rekursive Implementierung
2  gather([Pid | T]) ->
3    receive
4    % Ordnung der Ergeb. entspricht Ordnung der Argumente
5      { Pid, Ret } -> [Ret | gather(T)]
6    end;
7  gather([]) -> [].
```

- Die Ergebnisse werden in der Reihenfolge eingesammelt, in der die Prozesse erzeugt worden sind (da stets auf das Ergebnis der nächsten Pid in der in pmap erzeugten Liste von Prozess-IDs gewartet wird)
- Zeile 5 wartet, bis das Ergebnispaar (Pid, Ergebniswert) eintrifft
- Zeile 7 gibt eine leere Liste zurück wenn alle Prozess-ID geliefert haben
- Bemerkung: der Aufruf von gather(T) erfolgt nicht endrekursiv

Parallele Berechnung der Fibonacci-Zahlen

```
1  % Liste von Num Fibonacci-Zahlen
2  run(Num) ->
3    Seq = lists:seq(1, Num),          % Zufallszahlen erzeugen
4    Data = lists:map(fun(_) ->
5                 random:uniform(20) end, Seq),
6    % Berechnung parallel ausführen
7    pmap(fun fibo/1, Data).
```

- Die Funktion run(Num) erzeugt zuerst Num viele Zufallszahlen in einer Liste, die dann zusammen mit der Funktion fibo/1 an die Funktion pmap zur Berechnung des Ergebnisses übergeben wird

Diskussion

- Passende Abstraktion / Form der Verarbeitung wählen
  - Ist die Ordnung der Ergebnisse notwendig? (hier ja)
  - Werden Ergebnisse benötigt? (hier ja)
- Anzahl der parallelen Prozesse
  - Abhängig von Berechnungsmodell, Hardware etc.
  - Evtl. pmap mit max. Anzahl gleichzeitiger Prozesse implementieren
- Berechnungsaufwand der Prozesse
  - Trade-Off zwischen Aufwand für Berechnung vs. Aufwand für das Senden und Empfangen von Daten und Ergebnissen

Alternative Implementierung von pmap

- Falls do_fun() als Nachricht jeweils Tupel {Pid, {n, fibo(n)} liefern würde, dann kann ggf. auf die Ordnung der Ergebnismenge verzichtet werden
- Ein Zähler für bereits eingetroffene Ergebnisse hilft festzustellen, wenn alle Ergebnisse vorliegen

```
1  pmap(F, L) ->
2      ...
3      gather2(length(L), Ref, []).
4
5  gather2(N, Ref, L) ->
6      receive
7          {Ref, Ret } -> gather2(N-1, Ref, [Ret | L])
8      end;
9  gather2(0,_, L) -> L.
```

- Bereits vorliegende Ergebnisse können so direkt eingesammelt werden.

Diskussion des Speedup

- Eine Bestimmung des Speedup erfordert
  - Zeitmessung
  - Kontrolle der genutzten Prozessoren/Cores
- Welchen Einfluss hat die Anzahl der erzeugten Prozesse?

Speedup: Zeitmessung

- Nutzung der Funktion timer:tc/3

```
1> timer:tc(ch4_4, run, [30]).
{7900, [233,1,...]} % Liste enthält Ergebnisse
```

- Für bessere Reproduzierbarkeit sollte die Funktion mehrfach ausgeführt und dann der Durchschnitt der jeweiligen Laufzeiten berechnet werden (nach John Hughes: Programming in Erlang, www.cse.chalmers.se)

```
benchmark(M, Fun, D) ->
    % 100 Funktionsaufrufe
    Runs = [timer:tc(M, Fun, [D]) || _ <- lists:seq(1, 100)],
    % Durchschnitt der Laufzeiten in Millisekunden berechnen
    lists:sum([T || { T, _ } <- Runs]) / (1000 * length(Runs)).
```

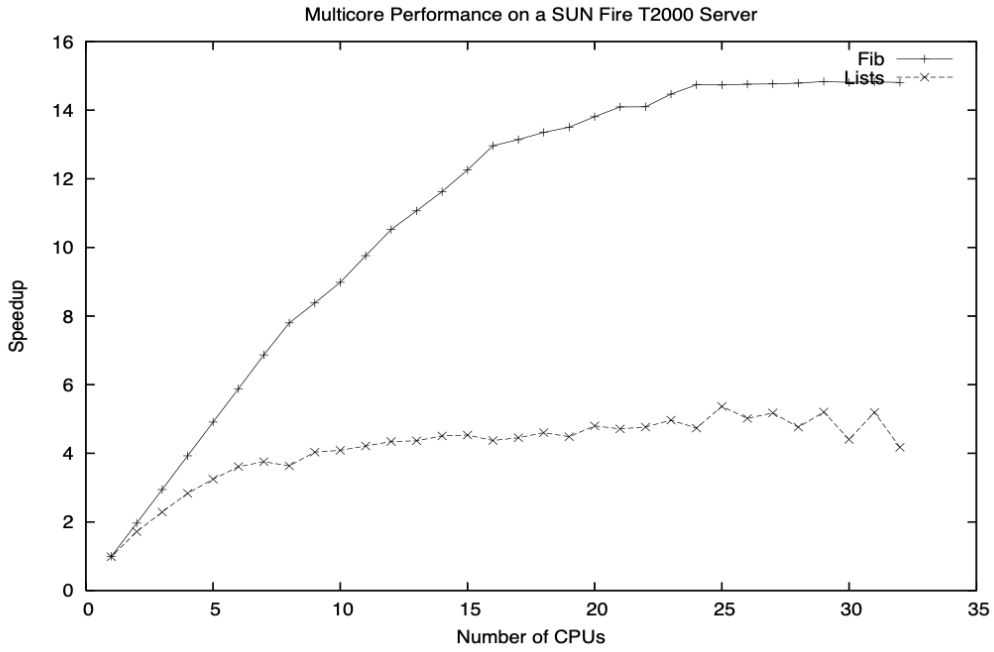

Bestimmung des Speedup

ch4_6:benchmark(ch4_4, run, 1000).

Anzahl Threads	Laufzeit (msecs.)
1	108
2	55
4	47
Sequentiell	108

- Achtung:
  - Der Aufwand für die Berechnung einer Fibonacci-Zahl ist nicht konstant
  - Es werden jeweils Zufallszahlen als Eingabe gegeben
  - Die Berechnungszeit unterschiedlicher Läufe ist daher unterschiedlich

Vergleich des Speedup bei unterschiedlichen Aufgaben



Vergleich des Speedup bei Sortieren von Listen vs. Fibonacci-Berechnung (Quelle: Armstrong. Programming Erlang: Software for a Concurrent World)

Datenparallelität: Das MapReduce-Paradigma

- Parallelisierungsmuster inspiriert von Konzepten funktionaler Programmiersprachen (map, reduce/fold)
- Basis von Big-Data-Plattformen wie Hadoop, Spark, …
- Grundidee:
  - map(F, Seq)
    - Wende die Funktion F (als Argument übergeben) auf alle Elemente einer Folge Seq an
    - Beispiel: Multipliziere jedes Element mit 2
  - reduce(F, Seq)
    - Wende eine Funktion F schrittweise auf die Elemente einer Folge Seq an und produziere einen einzelnen Wert
    - Beispiel: Berechne die Summe aller Elemente der Folge

Die Funktion map in Erlang

- Definition von `map` (auch als lists:map/2):

```
map(_, [])       -> [];
map(F, [H | T])  -> [F(H) | map(F, T)].
```

- Definition der Multiplikation:

```
mult(X)          -> X * 2.
```

- Anwendung:

```
1> S = [1,2,3,4].
[1,2,3,4]
2> mr:map(fun mr:mult/1, S).
[2,4,6,8]
```


Die Funktion reduce in Erlang

- Definition von `reduce` (auch als lists:foldl/3 bzw. lists:foldr/3):

```
reduce(_, Init, [])    -> Init;
map(F, Init, [H | T]) -> reduce(F, F(H, Init), T).
```

- Definition der Multiplikation:

```
add(X, Y) -> X + Y.
```

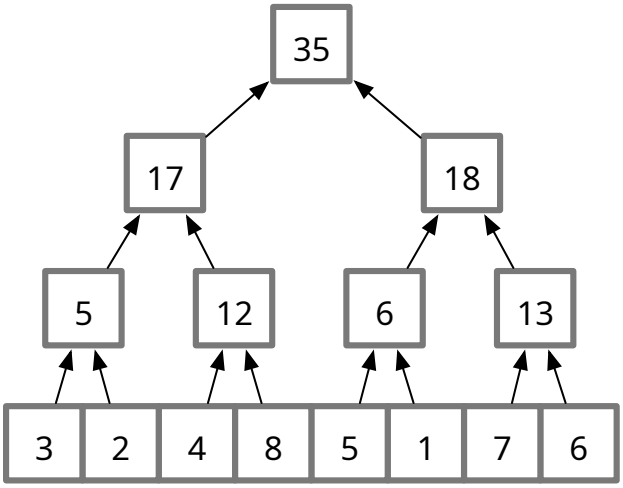
- Anwendung:

```
1> S = [1,2,3,4].
[1,2,3,4]
2> mr:reduce(fun mr:add/2, 0, S).
10
```


Parallelisierung von map und reduce in Erlang

- map
  - Die Funktion F kann unabhängig (parallel) auf jedes Element angewendet werden
  - Partitionieren und Verteilen der Elemente der Folge
  - Siehe pmap
- reduce
  - Ist die Funktion F assoziativ, kann ähnlich parallelisiert werden, d.h. die Funktion F kann auf Paare unabhängig angewendet werden
  - F assoziativ $:\Longleftrightarrow$ F(F(a, b), c) = F(a, F(b, c))
  - Achtung: manche Funktionen sind nur
    - linksassoziativ: a - b - c = (a - b) - c ≠ a - (b - c) bzw.
    - rechtsassoziativ: a ^ b ^ c = a ^ (b ^ c) ≠ (a ^ b) ^ c (Exponentiation)

Parallelisierung von reduce bei assoziativer Funktion



Beispiel: Die Funktion F(a, b) = a + b ist assoziativ, also kann nach diesem Schema parallelisiert werden.

Task-Parallelität: Sortieren (1)

- QuickSort in Erlang:

```
qsort([])      -> [];
qsort([H | T]) -> qsort([X | X <- T, X < H]) ++
                  [H] ++
                  qsort([Y | Y <- T, Y >= H]).
```

- Typische funktionale Notation von QuickSort mit List Comprehensions (siehe auch Kapitel 3a)
- Das erste Element H der noch zu sortierenden Teil-Liste dient jeweils als Pivot-Element
  - Achtung: führt bei schon sortierten Listen zu quadratischer Laufzeit!

Task-Parallelität: Sortieren (2)

- Idee:
  - Prozess für das Sortieren der einen Hälfte starten
  - Elternprozess kann die andere Hälfte sortieren
  - Rekursive Zerlegung ...

```
1 qsort2([])      -> [];
2 qsort2([H | T]) -> Parent = self(),
3                    spawn(fun() ->
4                          Parent ! qsort2([Y | Y <- T,
5                                           Y >= H]) end),
6                    qsort2([X | X <- T, X < H]) ++ [H] ++
7                     receive T2 -> T2 end.
```


Task-Parallelisierung: Sortieren (3)

- Erläuterungen:
  - Zeile 2 zerlegt rekursiv jede noch nicht leere Liste
  - Zeilen 3 - 5 erzeugen einen neuen Prozess zur Sortierung der Hälfte mit den Elementen, die größer als das Pivot-Element sind
  - Zeile 6 arbeitet wie bisher
  - Zeile 7 wartet auf den Empfang der sortierten Folge von dem in den Zeile 3 - 5 erzeugten Prozess
- Zeitmessung:

```
1> L = ch4_6:rand_list(100000).
...
2> ch4_6:benchmark(ch4_10, qsort, L).
131.90963
3> ch4_6:benchmark(ch4_10, qsort2, L).
293.59211
```


Task-Parallelisierung: Sortieren (4)

- Bewertung:
  - Diese erste parallele Version ist langsamer!
  - Mögliche Erklärung: Der Start neuer Prozesse ist aufwendiger als das Sortieren kleiner Teilfolgen
  - Bessere zweite parallele Version nach John Hughes: Parallel Programming in Erlang, www.cse.chalmers.se
    - Kontrolle der Granularität für parallele Ausführung
    - Danach: Sortieren mit der sequentiellen Variante
    - Einfache Begrenzung der parallelen Zerlegung
    - Weiterführende Überlegungen möglich, z.B. maximale Anzahl der entstehenden Prozesse so begrenzen, dass nicht mehr Prozesse erzeugt werden als CPU-Cores zur Verfügung stehen

Task-Parallelisierung: Sortieren (5)

```
1  qsort3(L) -> qsort3(4, L). % maximal 4 Rekursionsstufen parallel
2  qsort3(0, L) -> qsort(L).  % Umschalten auf sequentiell
3
4  qsort3(_, [])      -> [];
5  qsort3(N, [H | T]) -> Parent = self(),
6                        spawn(fun() ->
7                         Parent ! qsort3(N-1, [Y | Y <- T,
8                                               Y >= H]) end),
9                        qsort3(N-1, [X | X <- T, X < H]) ++
10                       [H] ++
11                       receive T2 -> T2 end.
```

```
4> ch4_6:benchmark(ch4_10, qsort3, L).
87.54315
```


Zusammenfassung

- Leichtgewichtige Prozesse als Baustein der Parallelisierung in Erlang
- Prozesskommunikation ausschließlich über Message Passing
- Der funktionale Charakter von Erlang (u.a. Vermeidung von Seiteneffekten) vereinfacht Parallelisierung deutlich
- Daten- und Taskparallelität möglich
- Hoher Abstraktionsgrad, aber auch wenig Einflussmöglichkeiten
- Der Overhead durch Prozesserzeugung und -verwaltung sollte beachtet und der Grad an Parallelisierung ggf. begrenzt werden