# In Erlang(en)
## Concurrency Oriented Programming

Jan Hermanns

# Motivation

- Carrier Class Systeme
  - massiv parallel
  - 99.9999999% Uptime
  - d.h. 31ms Downtime pro Jahr

Wie sieht eine Programmiersprache aus, mit der sich solche Anforderungen realisieren lassen?

Wir werden sehen ...

# Erlang OTP

- ursprünglich eine Plattform für Telekommunikationsanwendungen
  - mnesia DB
  - CORBA
  - ASN.1
  - yaws Webserver
  - Loadbalancer

# Erlang

- Deklarative syntax
  - funktionaler Sprachkern
  - weitgehend frei von Seiteneffekten
  - sehr kompakt (Faktor 4–10 weniger Code im Vergleich zu C/C++)
- Nebenläufigkeit eingebaut
  - leichtgewichtige Prozesse
  - asynchrones message passing
- Continuous operation
  - Code kann im laufenden System ausgetauscht werden

# Datentypen

- Primitive Datentypen
  - Integer (`42`, `63456345634563563`)
  - Float (`3.14159`)
  - Atome (`red`, `blue`, `'Atom mit Leerzeichen'`)
- Zusammengesetzte Datentypen
  - Tuppel (`{1,2}`, {person, {name, "Jan"}})
  - Listen (`[1,2,3]`, `[{id, 1}, 2, [3,4,5]]`)

Strings sind lediglich Listen von Integern!

```
>[74, 97, 110] == "Jan".
true
```

# Hello Erlang(en)

```
-module(ex0).
-export([say_hello/0]).

say_hello() -> io:format("Hello Erlangen\n").
```

```
> ex0:say_hello().
Hello Erlangen
```

# Einfache Funktionen

```
-module(ex1).
-export([square/1]).

square(X) -> times(X, X).

times(X, N) -> X * N.
```

```
> ex1:square(3).
9
> ex1:times(3,3).
** exited: {undef,[{ex1,times,[3,3]},
                   {erl_eval,expr,3},
                   {erl_eval,exprs,4},
                   {shell,eval_loop,2}]} **
```

# Funktionsdefinition mit Pattern-Matching

```
-module(ex2).
-export([area/1]).
-import(ex1, [square/1]).

area({square, Side})    -> square(Side);
area({rectangle, X, Y})-> X * Y;
area({circle, Radius}) -> 3.14159 * square(Radius).
```

```
> ex2:area({rectangle, 3, 4}).
12
> ex2:area({circle, 5}).
78.5397
```

# Der Match-Operator '='

- `Pattern = Expression`
  - Der Ausdruck `Expression` wird ausgewertet und gegen das Muster `Pattern` geprüft
  - Bei Erfolg werden alle Variablen im Muster `Pattern` an die jeweiligen Werte gebunden

```
> N = {2006, mathema}.
{2006,mathema}
> {A, B} = N.
{2006,mathema}
> A.
2006
> B.
mathema
```

# Einfache rekursive Funktionsdefinition

```
-module(ex3).
-export([factorial/1]).

factorial(0) -> 1;
factorial(N) -> N * factorial(N-1).
```

```
> ex3:factorial(6).
720
> ex3:factorial(42).
1405006117752879898543142606244511569936384000000000
```

# Tail-Rekursion

```
-module(ex4).
-export([factorial/1]).

factorial(N) -> factorial(N, 1).

factorial(0, A) -> A;
factorial(N, A) -> factorial(N-1, A*N).
```

Konstanter Speicherverbrauch
im Gegensatz zu ‘normaler’ Rekursion!

# Listen

```
> A = [].
[]
> B = [1|A].
[1]
> C = [2|B].
[2,1]
> D = [3|C].
[3,2,1]
> E = [a, b, c | D].
[a,b,c,3,2,1]
```

## rekursiv

```
len_rec([]) -> 0;
len_rec([_|T]) -> 1 + len_rec(T).

  len_rec([a,b,c]).
≈ 1 + len_rec([b,c]).
≈ 1 + 1 + len_rec([c]).
≈ 1 + 1 + 1 + len_rec([]).
≈ 1 + 1 + 1 + 0.
≈ 3.
```

## tail-rekursiv

```
len_tailrec([], A) -> A;
len_tailrec([_|T], A) -> len_tailrec(T, A+1).

  len_tailrec([a,b,c], 0).
≈ len_tailrec([b,c], 1).
≈ len_tailrec([c], 2).
≈ len_tailrec([], 3).
≈ 3.
```

# Binärer Suchbaum

```
insert(Key, Value, nil) -> {Key, Value, nil,nil};
insert(Key, Value, {Key, _, S, B}) ->
      {Key,Value,S,B};
insert(Key,Value, {Key1, V,S,B}) when Key<Key1 ->
      {Key1, V, insert(Key, Value, S), B};
insert(Key, Value, {Key1, V,S,B}) when Key>Key1->
      {Key1, V, S, insert(Key, Value, B)}.
```

```
> A = bintree:insert(mathema, 1, nil).
{mathema,1,nil,nil}
> B = bintree:insert(adam, 2, A).
{mathema,1,{adam,2,nil,nil},nil}
> C = bintree:insert(xavier, 3, B).
{mathema,1,{adam,2,nil,nil},{xavier,3,nil,nil}}
> D = bintree:insert(campus, 2006, C).
{mathema,1,{adam,2,nil,{campus,2006,nil,nil}},{xa
vier,3,nil,nil}}
```

# Binärer Suchbaum cont.

```
lookup(Key, nil) -> not_found;
lookup(Key, {Key,Value,_,_}) -> {found, Value};
lookup(Key, {Key1,_, Smaller,_}) when Key<Key1 ->
      lookup(Key, Smaller);
lookup(Key, {Key1,_,_,Bigger}) when Key > Key1 ->
      lookup(Key, Bigger).
```

```
> bintree:lookup(campus, D).
{found,2006}
> bintree:lookup(jan, D).
not_found
```

# weitere interessante Features

- List Comprehensions
  - `[ X*2 || X <- lists:seq(1,10), X>5].`
- Higher order functions
  - `lists:map(fun(X) -> X*2 end, [1,2,3]).`
- Records
  - `#person{age=31, name="Jan"}.`
- Bit-Syntax
  - `<<42, 2019:16>>`

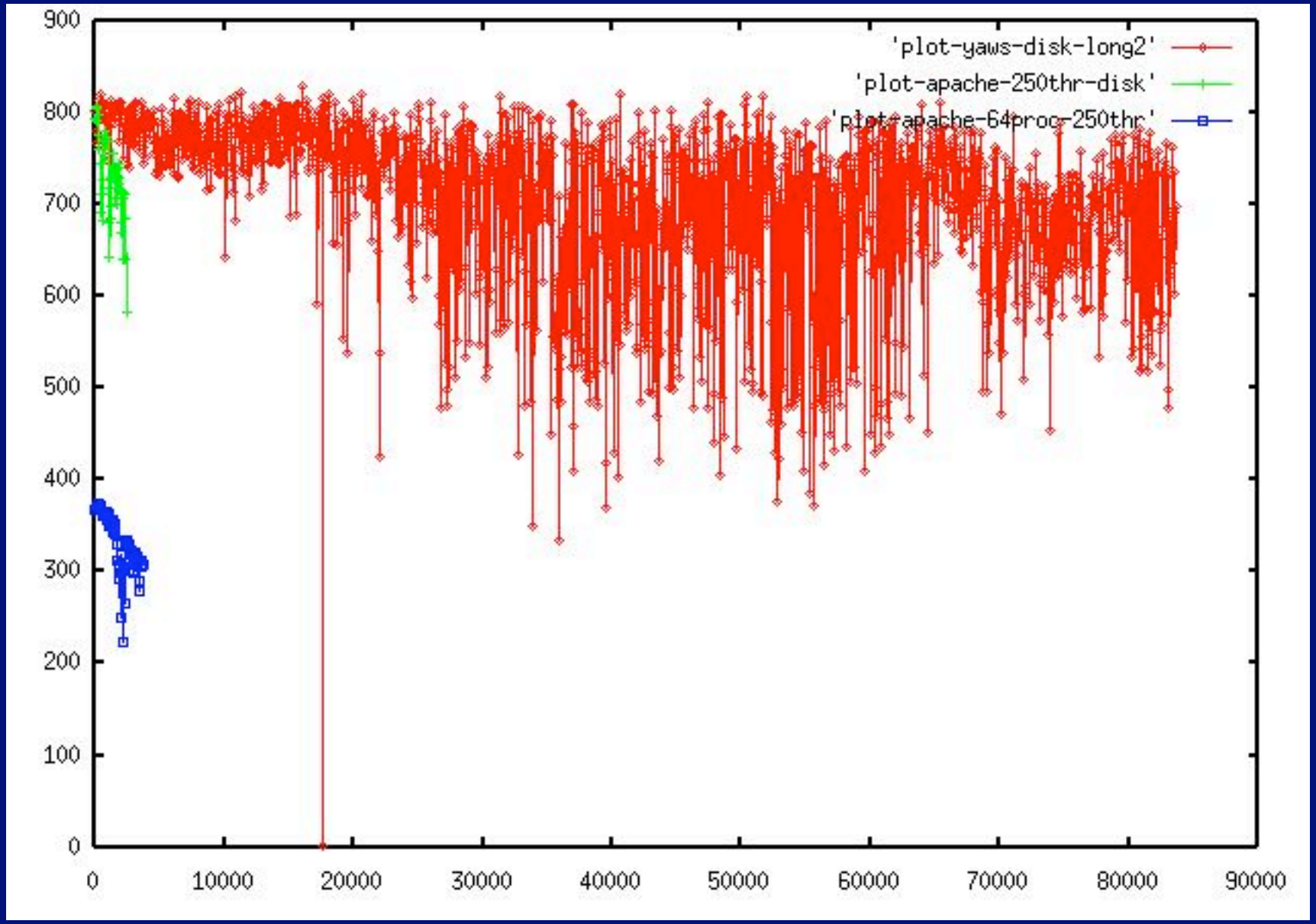
# Shared-Memory Probleme

- Programmabsturz in einer critical-region
- Programm verbringt zuviel Zeit in einer critical-region
- Deadlocks
- Locking ist i.d.R. sehr grob granular
- Shared-Memory skaliert schlecht

# Concurrency, the Erlang way

- Leichtgewichtige Prozesse (keine Betriebssystem Prozesse oder Threads)
- asynchrones Message-Passing (kein Shared-Memory, keine Locks/Semaphoren/Mutexes)

# Apache vs. Yaws



# Concurrency-Funktionen und Sprachkonstrukte

## Prozess erzeugen

```
Pid = spawn(Module, Function, ArgumentList)
```

## Nachricht senden

```
Pid ! Message
```

## Nachricht empfangen

```
receive     Message1 -> Action1;
            Message2 -> Action2;
            ...
after       Timeout -> TimeoutAction
end
```

# counter-example ;-)

```
-module(counter).
-export([start/0, loop/1]).

start() -> spawn(counter, loop, [0]).

loop(Value) -> receive
                 increment -> loop(Value + 1)
               end.
```

```
> Pid = counter:start().
<0.44.0>
> Pid ! increment.
increment
```

# Einfache Concurrency Patterns

Triggern: `A ! B`

Event: `receive A -> A end`

RPC-Call:

```
A ! {self(), B},
receive
  {A, Reply} -> Reply
end
```

Callback:

```
receive
  {From, A} -> From ! F(A)
end
```

```
-module(counter).
-export([start/0, increment/1, get_value/1, loop/1]).

start() -> spawn(counter, loop, [0]).

increment(Pid) -> Pid ! increment.

get_value(Pid) -> Pid ! {self(), getvalue},
                  receive
                      Value -> Value
                  end.

loop(Value) ->
  receive
      increment          -> loop(Value + 1);
      {From, getvalue}   -> From ! Value,
                            loop(Value);
      Otherwise          -> loop(Value)
  end.
```

# Resourcen

- Erlang
  - http://www.erlang.org/
  - http://www.erlang.se/
  - http://www.planeterlang.org/
  - http://www.trapexit.org/
  - http://armstrongonsoftware.blogspot.com/
  - http://www.sics.se/~joe/index.html
- Purely Functional Data Structures
  - http://www.cs.cmu.edu/~rwh/theses/okasaki.pdf