Parallel Programming in Erlang

John Hughes

What is Erlang?

Erlang $\simeq$ Haskell
- Types
- Lazyness
- Purity
+ Concurrency
+ Syntax

If you know Haskell, Erlang is easy to learn!

QuickSort again

- Haskell

```
qsort [] = []
qsort (x:xs) = qsort [y | y <- xs, y<x]
               ++ [x]
               ++ qsort [y | y <- xs, y>=x]
```

- Erlang

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

qsort [] =

- Haskell

```
qsort [] = []
qsort (x:xs) = qsort [y | y <- xs, y<x]
               ++ [x]
```

qsort([]) ->

- Erlang

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

QuickSort again

- Haskell

```
qsort [] = []
qsort (x:xs) = q                   <- xs
               ++ [x]
               ++ qsort [y | y <- xs
```

;

.

- Erlang

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

Qu

x:xs

gain

- Haskell

```
qsort [] = []
qsort (x:xs) = qsort [y | y <- xs, y<x]
               ++ [x]
               ++                     xs, y>=x]
```

[X|Xs]

- Erlang

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

QuickSort again

- Haskell

```
qsort [] = []
qsort (x:xs) = qsort [y | y <- xs, y<x]
               ++ [x]
               ++ qsort [y | y <- xs
```

- Erlang

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

foo.erl

Declare the module name

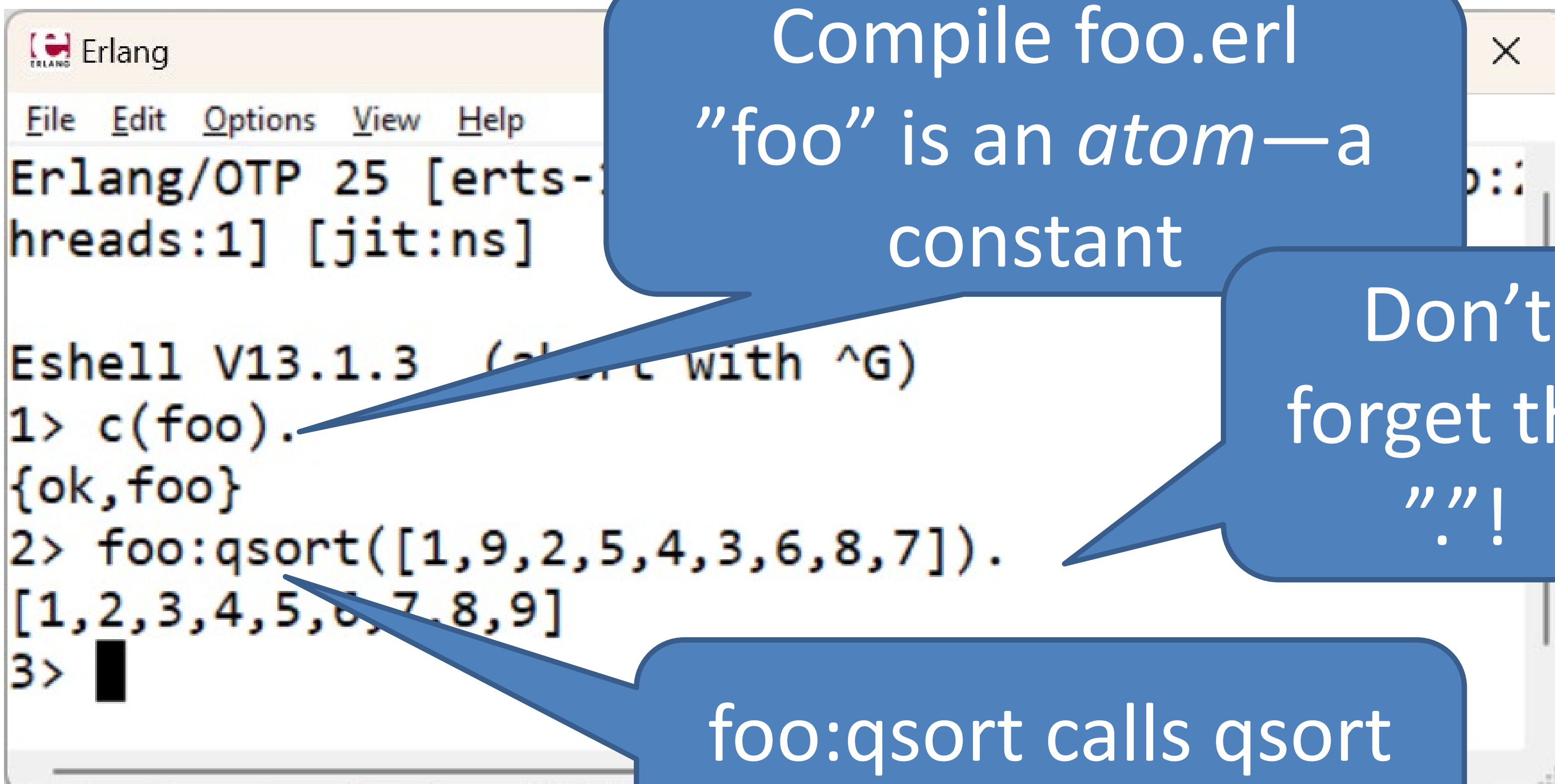
```
-module(foo).
-compile(export_all).

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

Simplest just to export everything

```
-compile([export_all,nowarn_export_all]).
```

werl/erl REPL



Compile foo.erl
"foo" is an *atom*—a constant

Don't forget the "."!

foo:qsort calls qsort from the foo module

- Much like ghci

Test Data

- Create some test data; in foo.erl:

```
random_list(N) ->
    [rand:uniform(1000000) || _ <- lists:seq(1,N)].
```

Side-effects!

Instead of [1..N]

- In the

```
L = foo:random_list(200000).
```

Timing calls

Module

Function

Arguments

```
4> timer:tc(foo,qsort,[L]).
{84172,
 [1,5,7,10,23,24,25,29
58,3,61,62,62,62,67,6
  7 81,94,96,102,105,106|...]}
```

atoms—i.e. constants

Microseconds

{A,B,C} is a tuple

Benchmarking

Binding a name... c.f. **let**

Macro: current module name

```
benchmark(Fun,L) ->
    NRuns = 500,
    {T,_} = timer:tc(fun()->
                [?MODULE:Fun(L)
                || _ <- lists:seq(1,NRuns)],
                ok
              end),
    T / (1000*NRuns).
```

```
4> foo:benchmark(qsort,L).
74.653528
```

Parallelism

```
68> erlang:system_info(schedulers_online).
4
```

4 OS threads! Let's use them!

Parallelism in Erlang

- Processes are created *explicitly*

```
Pid = spawn_link(fun() -> …Body… end)
```

- Start a process which executes …Body…
- `fun() -> Body end` ~ `\() -> Body`
- `Pid` is the *process identifier*

<0.86.0>

Parallel Sorting

```
psort([]) ->
    [];
psort([X|Xs]) ->
    spawn_link(
      fun() ->
        psort([Y || Y <- Xs, Y >= X])
      end),
    psort([Y || Y <- Xs, Y < X]) ++
          [X] ++
          ???.
```

Sort second half in parallel…

But how do we get the result?

Message Passing

Pid ! Msg

- Send a message to Pid
- *Asynchronous*—do not wait for delivery

Message Receipt

```
receive
    Msg -> …
end
```

- Wait for a message, then bind it to Msg

Parallel Sorting

```
psort([]) ->
    [];
psort([X|Xs]) ->
    Parent = self(),
    spawn_link(
      fun() ->
        Parent !
            psort([Y || Y <- Xs, Y >= X])
      end),
    psort([Y || Y <- Xs, Y < X]) ++
          [X] ++
          receive Ys -> Ys end.
```

The Pid of the executing process

Send the result back to the parent

Wait for the result *after* sorting the first half

Benchmarks

```
69> foo:benchmark(qsort,L).
74.318884
70> foo:benchmark(psort,L).
154.869018
```

- Parallel sort is slower! *Why?*

Controlling Granularity

```
psort2(Xs) -> psort2(5,Xs).

psort2(0,Xs) -> qsort(Xs);
psort2(_,[]) -> [];
psort2(D,[X|Xs]) ->
    Parent = self(),
    spawn_link(fun() ->
      Parent !
        psort2(D-1,[Y || Y <- Xs, Y >= X])
       end),
    psort2(D-1,[Y || Y <- Xs, Y < X]) ++
    [X] ++
    receive Ys -> Ys end.
```

Benchmarks

```
69> foo:benchmark(qsort,L).
74.318884
70> foo:benchmark(psort,L).
154.869018
71> foo:benchmark(psort2,L).
35.882206
```

- >2x speedup

Correctness

```
91> foo:psort2(L) == foo:qsort(L).
false
92> foo:psort2("hello world").
" edhllloorw"
```

Oops!

What’s going on?

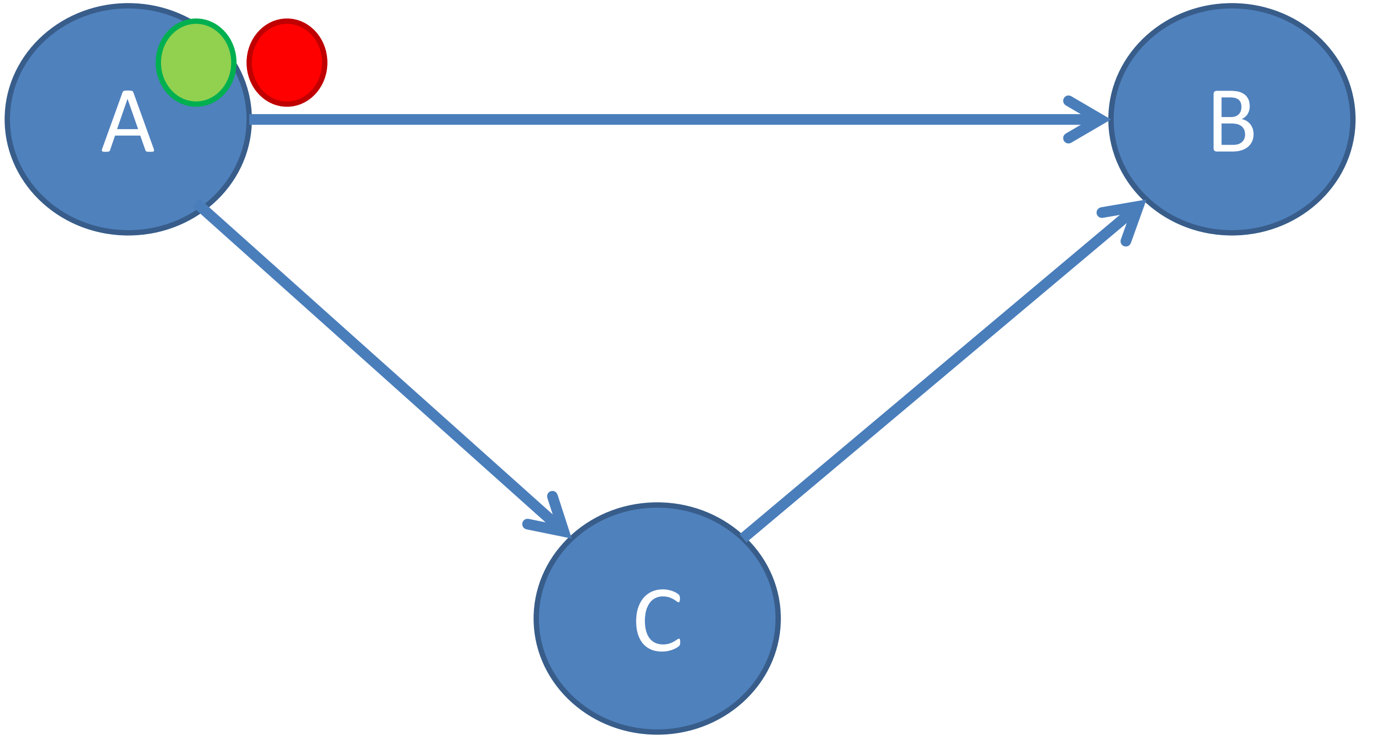
```
psort2(D,[X|Xs]) ->
    Parent = self(),
    spawn_link(fun() ->
      Parent ! …
       end),
    psort2(D-1,[Y || Y <- Xs, Y < X]) ++
    [X] ++
    receive Ys -> Ys end.
```

What's going on?

```
psort2(D,[X|Xs]) ->
    Parent = self(),
    spawn_link(fun() ->
      Parent ! …
      end),
    Parent = self(),
    spawn_link(fun() ->
      Parent ! …
      end),
    psort2(D-2,[Y || Y <- Xs, Y < X]) ++
    [X] ++
    receive Ys -> Ys end ++
    [X] ++
    receive Ys -> Ys end.
```

Message Passing Guarantees

Message Passing Guarantees



Tagging Messages Uniquely

```
Ref = make_ref()
```

- Create a globally unique reference

```
Parent ! {Ref,Msg}
```

- Send the message tagged with the reference

```
receive {Ref,Msg} -> ... end
```

- Match the reference on receipt… picks the right message from the mailbox

A correct parallel sort

```
psort3(Xs) ->
    psort3(5,Xs).

psort3(0,Xs) ->
    qsort(Xs);
psort3(_,[]) ->
    [];
psort3(D,[X|Xs]) ->
    Parent = self(),
    Ref = make_ref(),
    spawn_link(fun() ->
      Parent ! {Ref,psort3(D-1,[Y || Y <- Xs, Y >= X])}
    end),
    psort3(D-1,[Y || Y <- Xs, Y < X]) ++
      [X] ++
      receive {Ref,Greater} -> Greater end.
```

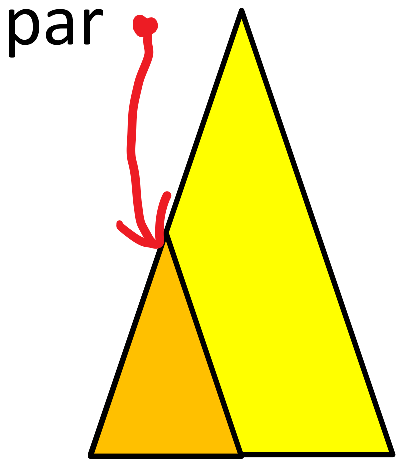
Tests

```
69> foo:benchmark(qsort,L).
74.318884
…
72> foo:benchmark(psort3,L).
36.818348
73> foo:qsort(L) == foo:psort3(L).
true
```

- >2x speedup, and now it works ☺

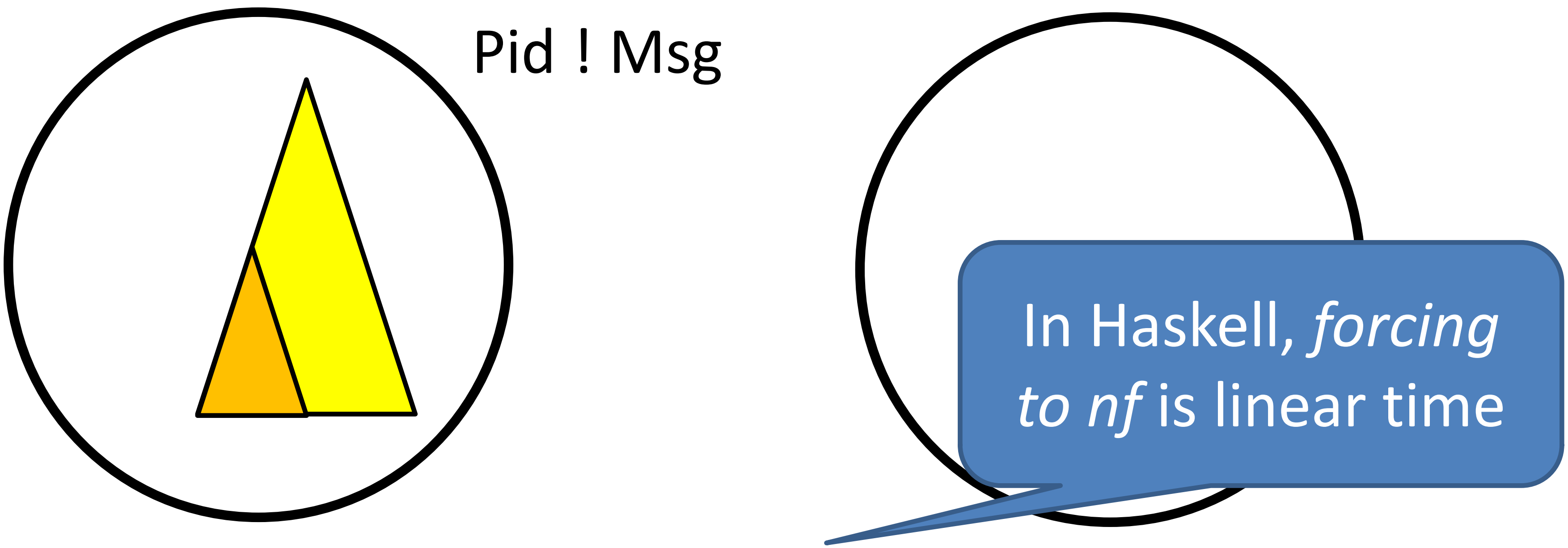
Parallelism in Erlang vs Haskell

- Haskell processes *share memory*



Parallelism in Erlang vs Haskell

- Erlang processes each have their own heap



- Messages have to be *copied*
- No global garbage collection—each process collects its own heap

What’s copied here?

```
psort3(D,[X|Xs]) ->
    Parent = self(),
    Ref = make_ref(),
    spawn_link(fun() ->
      Parent !
        {Ref,
         psort3(D-1,[Y || Y <- Xs, Y >= X])}
    end),
```

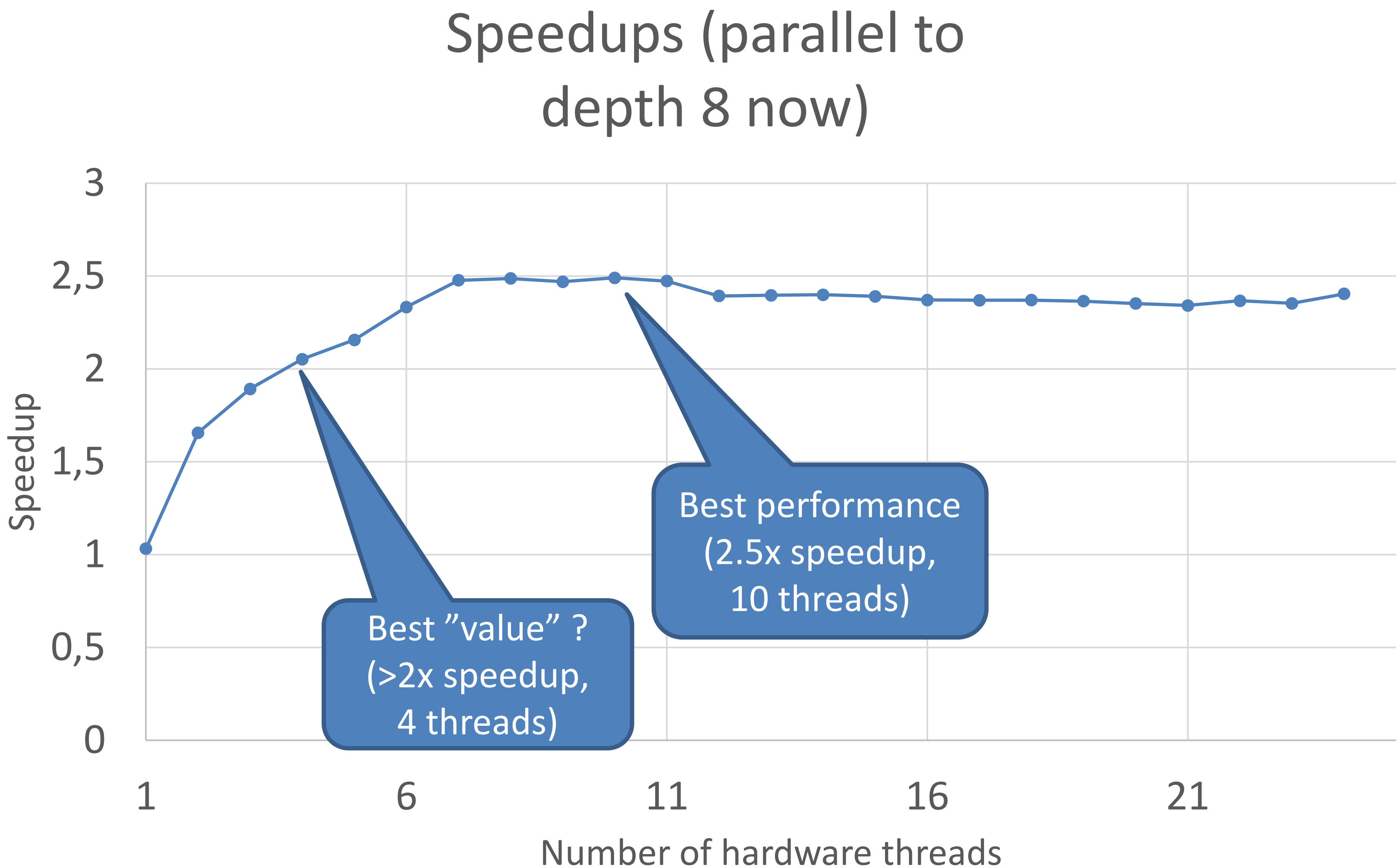
- Is it sensible to copy *all of Xs* to the new process?

Better

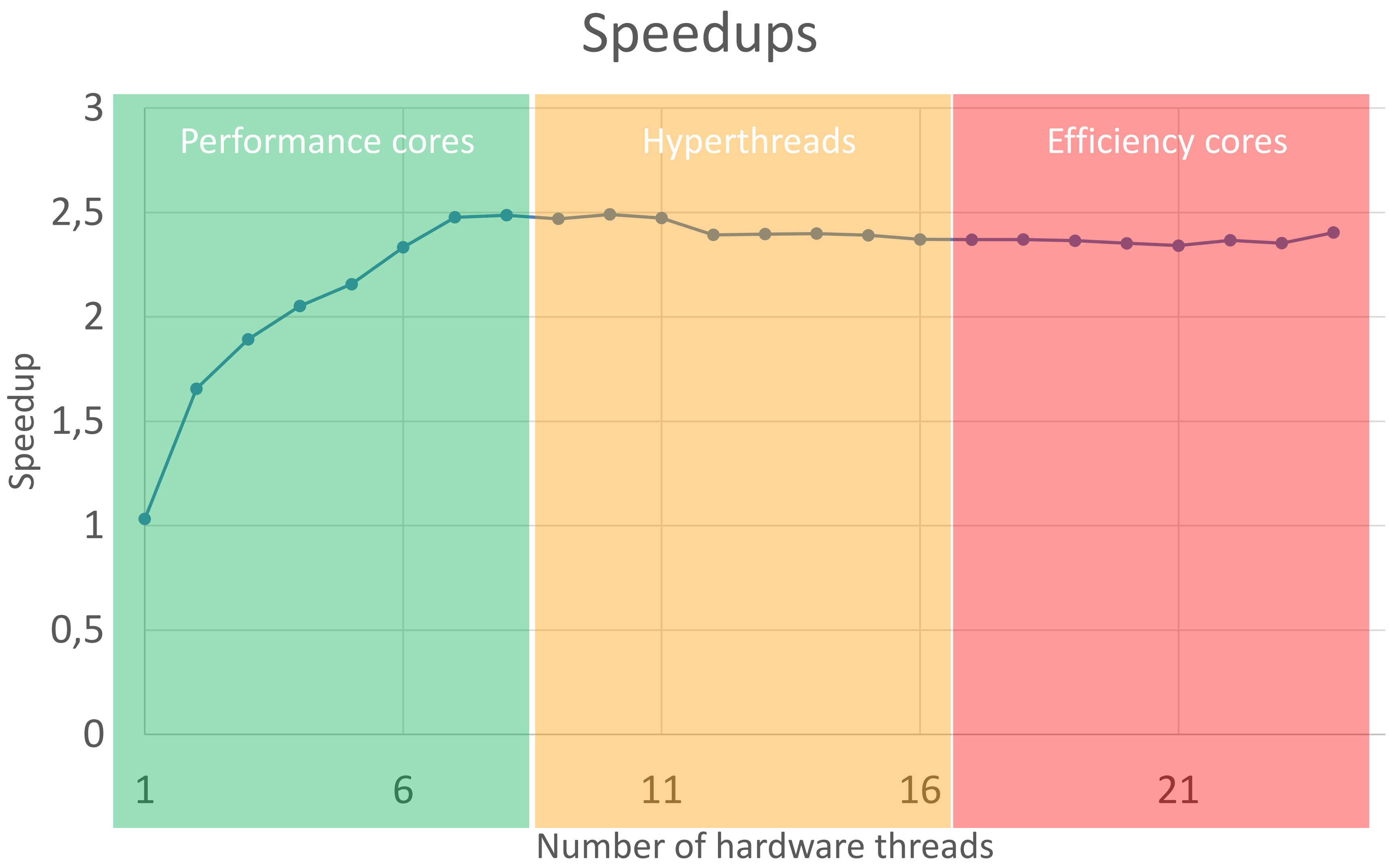
A small improvement—but Erlang lets us *reason* about copying

```
psort4(D,[X|Xs]) ->
    Parent = self(),
    Ref = make_ref(),
    Grtr = [Y || Y <- Xs, Y >= X],
    spawn_link(fun() ->
        Parent ! {Ref,psort4(D-1,Grtr)}
    end),
```

```
72> foo:benchmark(psort3,L).
36.818348
…
74> foo:benchmark(psort4,L).
30.543886
```



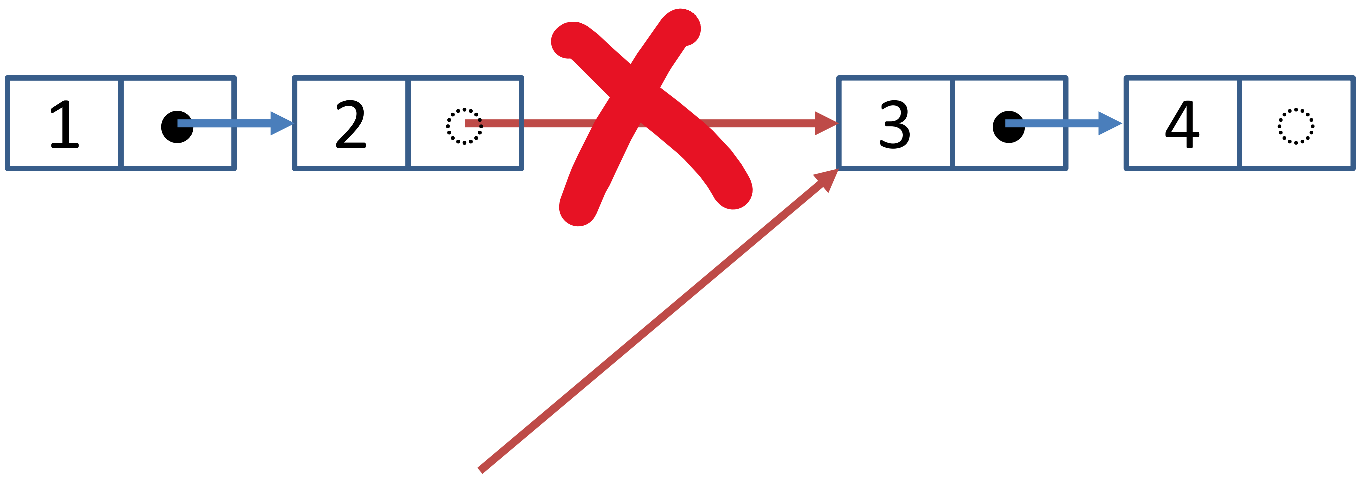
Speedups (parallel to depth 8 now)
3
2,5
2
1,5
1
0,5
0
Speedup
1
6
11
16
21
Number of hardware threads
Best ”value” ? (>2x speedup, 4 threads)
Best performance (2.5x speedup, 10 threads)



Core i9, 16 cores, L1 cache 1.4MB *in total*, L2 cache 14MB, L3 cache 30MB
(Data size for this problem: 200K x 16B = 3.2MB)

Reducing copying

[1,2] ++ [3,4] == [1,2,3,4]



Recall qsort

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

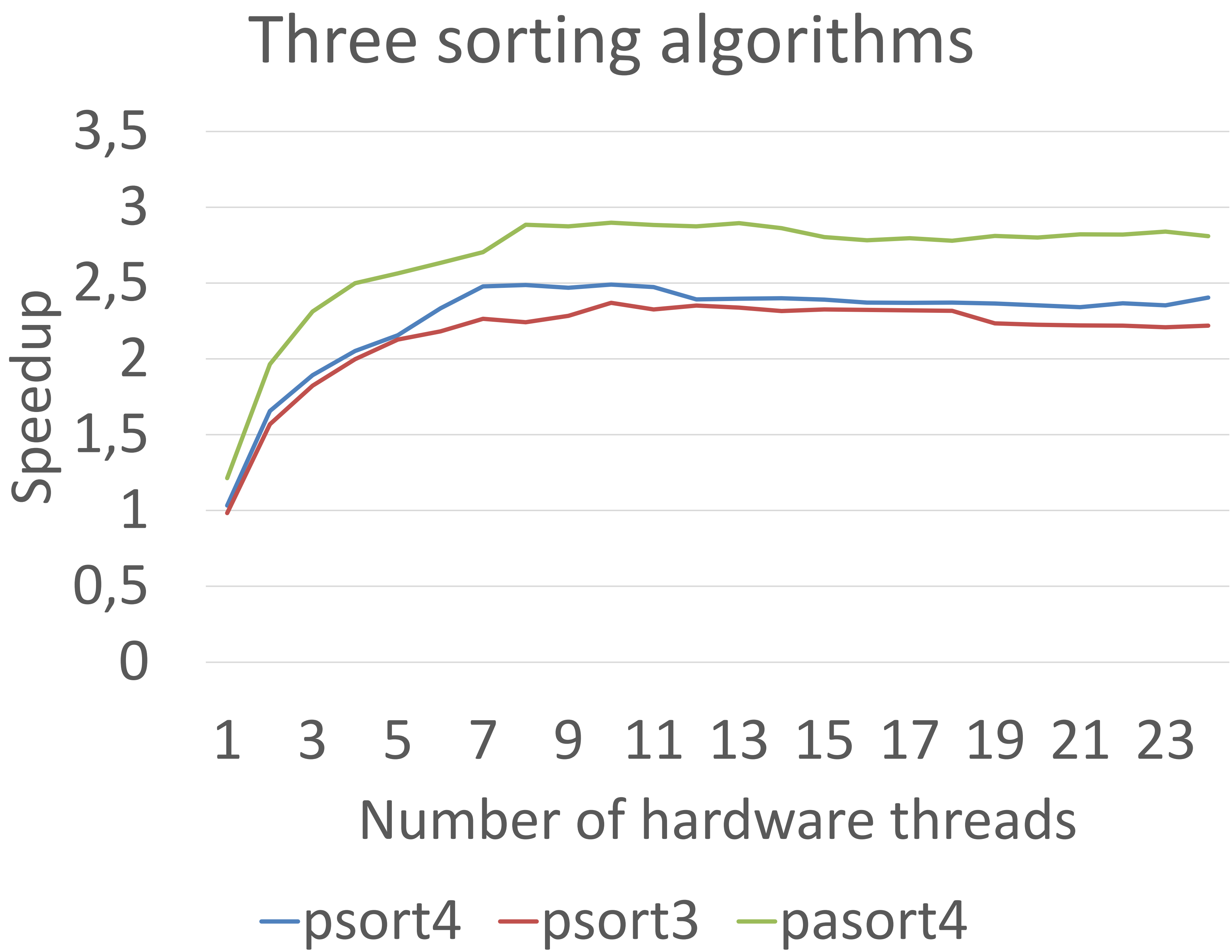
Reducing copying with an *accumulating parameter*

- Define asort s.t. asort(L,Acc) == qsort(L) ++ Acc

```
asort(L) -> asort(L, []).

asort([], Acc) -> Acc;
asort([X|Xs],Acc) ->
  asort([Y || Y <- Xs, Y<X],
        [X | asort([Y || Y <- Xs, Y>=X], Acc)]).
```

```
24> foo:benchmark(qsort,L).
79.116342
25> foo:benchmark(asort,L).
64.204434
```


Three sorting algorithms
Speedup
0
0,5
1
1,5
2
2,5
3
3,5
1
3
5
7
9
11
13
15
17
19
21
23
Number of hardware threads
psort4
psort3
pasort4

Haskell vs Erlang

- Sorting (different) random lists of 200K integers, on 2-core i7

	Haskell	Erlang
Sequential sort	353 ms	312 ms
Depth 5 //el sort	250 ms	153 ms

Erlang scales better

- *Despite* Erlang running on a VM!

Erlang Distribution

- Erlang processes can run on *different machines* with the same semantics

- No shared memory between processes!

- Just a little slower to communicate…

Named Nodes

```
werl -sname baz
```

- Start a node with a *name*

```
(baz@HALL)1> node().
baz@HALL
```

Node name is an atom

```
(baz@HALL)2> nodes().
[]
```

List of connected nodes

Connecting to another node

```
net_adm:ping(Node).
```

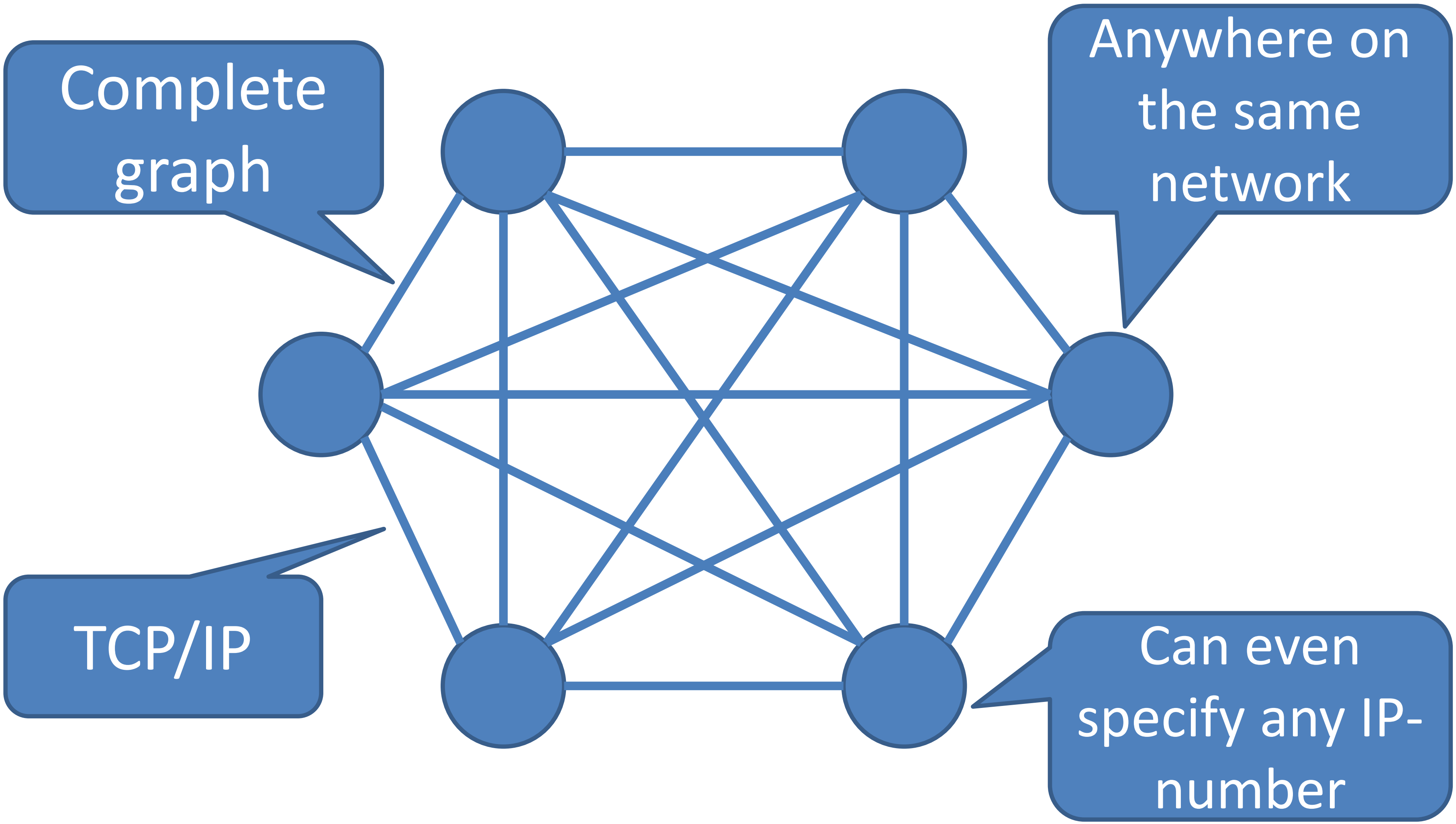
```
3> net_adm:ping(foo@HALL).
pong
```

Success—pang means connection failed

```
4> nodes().
[foo@HALL,baz@JohnsTablet2014]
```

Now connected to foo and other nodes foo knows of

Node connections



Gotcha! the Magic Cookie

- All communicating nodes must share the same *magic cookie* (an atom)

- Must be the same on all machines
  - By default, randomly generated on each machine

- Put it in $HOME/.erlang.cookie
  - E.g. cookie

A Distributed Sort

```
dsort([]) ->
    [];
dsort([X|Xs]) ->
    Parent = self(),
    Ref = make_ref(),
    Grtr = [Y || Y <- Xs, Y >= X],
    spawn_link(baz@JohnsDesktop,
       fun() ->
            Parent ! {Ref,psort4(Grtr)}
       end),
    psort4([Y || Y <- Xs, Y < X]) ++
     [X] ++
     receive {Ref,Greater} -> Greater
end.
```

Benchmarks

```
5> foo:benchmark(psort4,L).
87.23
6> foo:benchmark(dsort,L).
109.27
```

- Distributed sort is *slower*
  - Communicating between nodes is slower
  - Nodes on the same machine are sharing the cores anyway!

OK…

A 2-core laptop… silly to send it half the work

```
dsort2([X|Xs]) ->
    …
    spawn_link(baz@JohnsTablet2014,
      fun() ->
          ….
```

```
5> foo:benchmark(psort4,L).
87.23
6> foo:benchmark(dsort,L).
109.27
7> foo:benchmark(dsort2,L).
1190.33
```

Distribution Strategy

- Divide the work into 32 chunks on the master node
- Send *one chunk at a time* to each node for sorting
  - Slow nodes will get fewer chunks
- Use the fast parallel sort on each node

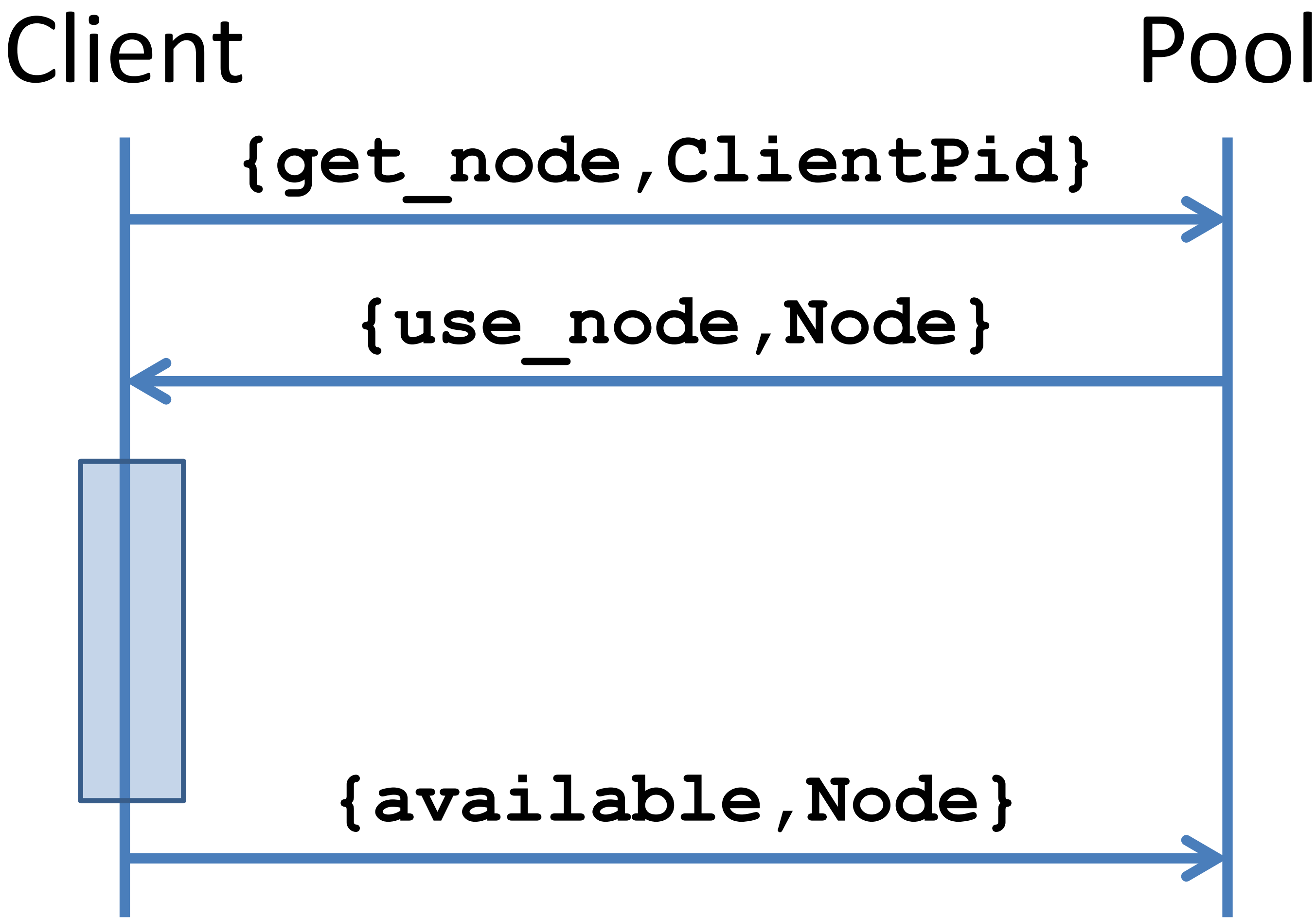
Node Pool

- We need a pool of *available nodes*

```
pool() ->
    Nodes = [node()|nodes()],
    spawn_link(fun() ->
      pool(Nodes)
    end).
```

- We create a process to manage the pool, initially containing all the nodes

Node Pool Protocol



Node Pool Behaviour

```
pool([]) ->
   receive
      {available,Node} ->
          pool([Node])
   end;
pool([Node|Nodes]) ->
   receive
      {get_node,Pid} ->
          Pid ! {use_node,Node},
          pool(Nodes)
   end.
```

If the pool is empty, wait for a node to become

If nodes are available, wait for a request and give one out

Selective receive is really useful!

dwsort

```
dwsort(Xs) -> dwsort(pool(),5,Xs).

dwsort(_,_,[]) -> [];
dwsort(Pool,D,[X|Xs]) when D > 0 ->
    Grtr = [Y || Y <- Xs, Y >= X],
    Ref = make_ref(),
    Parent = self(),
    spawn_link(fun() ->
      Parent ! {Ref,dwsort(Pool,D-1,Grtr)}
    end),
    dwsort(Pool,D-1,[Y || Y <- Xs, Y < X]) ++
      [X] ++
      receive {Ref,Greater} -> Greater end;
```

Parallel recursion to depth 5

dwsort

```
dwsort(Pool,0,Xs) ->
    Pool ! {get_node,self()},
    receive
      {use_node,Node} ->
         Ref = make_ref(),
         Parent = self(),
         spawn_link(Node, fun() ->
            Ys = psort4(Xs),
            Pool ! {available,Node},
            Parent ! {Ref,Ys}
         end),
         receive {Ref,Ys} -> Ys end
    end.
```

A further optimisation: if we should use the *current* node, don't spawn a new process

Messages between nodes

Benchmarks

```
(baz@HALL)17> foo:benchmark(qsort,L).
271.97
(baz@HALL)18> foo:benchmark(psort4,L).
88.65
(baz@HALL)19> foo:benchmark(dsort2,L).
1190.33
(baz@HALL)20> nodes().
[baz@JohnsTablet2014]
(baz@HALL)21> foo:benchmark(dwsort,L).
295.59
(baz@HALL)22> foo:benchmark(dwsort2,L).
195.05
```

With each node in the pool *twice*, to overlap communication and computatíon

Oh well!

- It’s quicker to *sort* a list, than to send it to another node and back!

Another Gotcha!

- All the nodes must be running *the same code*
  - Otherwise sending functions to other nodes cannot work

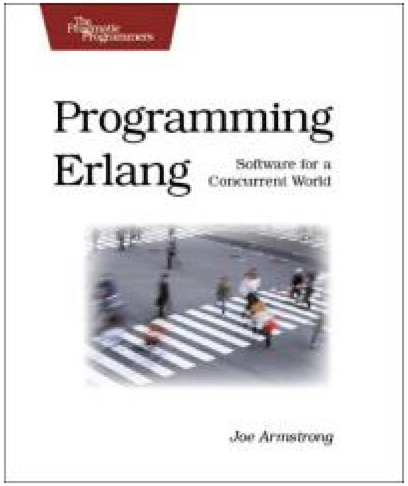
- **`nl(Mod)`** loads the module on *all* connected nodes.

Summary

- Erlang parallelism is more explicit than in Haskell
- Processes do not share memory
- All communication is explicit by message passing
- Performance and scalability are strong points
- Distribution is easy
  - (But sorting is cheaper to do than to distribute☹)

References

- *Programming Erlang: Software for a Concurrent World*, Joe Armstrong, Pragmatic Bookshelf, 2007.



- *Learn you some Erlang for Great Good*, Frederic Trottier-Hebert , http://learnyousomeerlang.com/