Parallel Programming in Erlang (PFP Lecture 10)

John Hughes
What is Erlang?

Erlang

- Haskell
- Types
- Lazyness
- Purity
+ Concurrency
+ Syntax

If you know Haskell, Erlang is easy to learn!
QuickSort again

• Haskell

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

• Erlang

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

• 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]).
```
QuickSort again

• Haskell

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

• Erlang

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

- Haskell

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

- Erlang

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

• Haskell

\[
\begin{align*}
\text{qsort} & \ [\ ] = [\ ] \\
\text{qsort} & \ (x:xs) = \text{qsort} \ [y \mid y \leftarrow xs, \ y < x] \\
& \quad \quad \quad \quad + [x] \\
& \quad \quad \quad \quad + \text{qsort} \ [y \mid y \leftarrow xs, \ y \geq x].
\end{align*}
\]

• Erlang

\[
\begin{align*}
\text{qsort}([\ ]) & \rightarrow [\ ]; \\
\text{qsort}([X|Xs]) & \rightarrow \text{qsort}([Y \mid Y \leftarrow Xs, \ Y < X]) \\
& \quad \quad \quad \quad + [X] \\
& \quad \quad \quad \quad + \text{qsort}([Y \mid Y \leftarrow Xs, \ Y \geq X]).
\end{align*}
\]
foo.erl

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

qsort([]) ->
    [];
qsort([X|Xs]) ->
    qsort([Y || Y <- Xs, Y<X]) ++
    [X] ++
    qsort([Y || Y <- Xs, Y>=X]).
 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) ->
    [random:uniform(1000000) || _ <- lists:seq(1,N)].

• In the shell:

L = foo:random_list(200000).

Side-effects!

Instead of [1..N]
Timing calls

### Microseconds

<table>
<thead>
<tr>
<th>Module</th>
<th>Function</th>
<th>Arguments</th>
<th>Atoms — i.e. constants</th>
<th>Tuple</th>
</tr>
</thead>
<tbody>
<tr>
<td>timer:tc</td>
<td>foo, qsort</td>
<td>[L]</td>
<td>{390000, [1, 2, 6, 8, 11, 21, 33, 37, 69, 61, 69, 70, 75, 86, 91, 105, 106, 112, 117, 118, 123, ...]}</td>
<td>{A, B, C} is a tuple</td>
</tr>
</tbody>
</table>
BENCHMARKING

- 100 runs, average & convert to ms

80> foo:benchmark(qsort,L).
330.88
Parallelism

34> erlang:system_info(schedulers).

4

Four OS threads! Let’s use them!
Parallelism in Erlang

- Processes are created *explicitly*

\[
\text{Pid} = \text{spawn\_link}(\text{fun}() \rightarrow \ldots\text{Body}\ldots\ \text{end})
\]

- Start a process which executes \...\text{Body}...
- \text{fun}(\) \rightarrow \text{Body\ end} \sim \text{\}() \rightarrow \text{Body}
- \text{Pid} is the \text{process identifier}
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

84> foo:benchmark(qsort,L).
327.13
85> foo:benchmark(psort,L).
474.43

• Parallel sort is slower! Why?
Controlling Granularity

\[
\text{psort2}(Xs) \rightarrow \text{psort2}(5,Xs).
\]

\[
\text{psort2}(0,Xs) \rightarrow \text{qsort}(Xs);
\]

\[
\text{psort2}(_,[]) \rightarrow [];
\]

\[
\text{psort2}(D,[X|Xs]) \rightarrow
\]

\[
\text{Parent} = \text{self}(),
\]

\[
\text{spawn_link}(\text{fun}() \rightarrow
\]

\[
\text{Parent} !
\]

\[
\text{psort2}(D-1,[Y || Y \leftarrow Xs, Y \geq X])
\]

\[
\text{end},
\]

\[
\text{psort2}(D-1,[Y || Y \leftarrow Xs, Y < X]) ++
\]

\[
[X] ++
\]

\[
\text{receive Ys} \rightarrow \text{Ys end}.
\]
Benchmarks

• 2x speedup on 2 cores (x2 hyperthreads)
Profiling Parallelism with Percept

File to store profiling information in

87> percept:profile("test.dat",{foo,psort2,[L]},[procs]).
Starting profiling.
ok
Profiling Parallelism with Percept

88> percept:analyze("test.dat").
Parsing: "test.dat"
Consolidating...
Parsed 160 entries in 0.078 s.
   32 created processes.
   0 opened ports.
ok

Analyse the file, building a RAM database
Profiling Parallelism with Percept

Start a web server to display the profile on this port

90> percept:start_webserver(8080).
{started,"JohnsTablet2012",8080}
Profiling Parallelism with Percept

Shows runnable processes at each point

4 procs
Profiling Parallelism with Percept
Correctness

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

Oops!
What’s going on?

```plaintext
code
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?

```plaintext
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

23> foo:benchmark(qsort,L).
329.48
24> foo:benchmark(psort3,L).
166.66
25> foo:qsort(L) == foo:psort3(L).
true

• Still a 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
  Pid ! Msg

• Messages have to be copied

• No global garbage collection—each process collects its own heap

In Haskell, forcing to nf is linear time
What’s copied here?

```haskell
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?
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),

31> foo:benchmark(psort3,L).
166.3
32> foo:benchmark(psort4,L).
152.6
Benchmarks on 4 core i7

17> foo:benchmark(qsort,L).
    414.07
18> foo:benchmark(psort4,L).
    144.8

• Speedup: 2.9x on 4 cores/8 threads
  – (increased depth to 8)
Haskell vs Erlang

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

<table>
<thead>
<tr>
<th></th>
<th>Haskell</th>
<th>Erlang</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential sort</td>
<td>353 ms</td>
<td>312 ms</td>
</tr>
<tr>
<td>Depth 5 //el sort</td>
<td>250 ms</td>
<td>153 ms</td>
</tr>
</tbody>
</table>

- *Despite* Erlang running on a VM!

Erlang scales much better
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**

- Start a node with a *name*

  
  (baz@JohnsTablet2012)1> node().

  baz@JohnsTablet2012

  (baz@JohnsTablet2012)2> nodes().

  []

  **Node name is an atom**

  **List of connected nodes**
Connecting to another node

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

```
3> net_adm:ping(foo@JohnsTablet2012).
pong
```

```
4> nodes().
[foo@JohnsTablet2012,baz@HALL]
```

Success—pang means connection failed

Now connected to foo and other nodes foo knows of
Node connections

Complete graph

Anywhere on the same network

TCP/IP

Can even specify any IP-number
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

dsорт([]) ->
    [];
dsорт([X|Xs]) ->
    Parent = self(),
    Ref = make_ref(),
    Grtr = [Y || Y <- Xs, Y >= X],
    spawn_link(foo@JohnsTablet2012, fun() ->
        Parent ! {Ref,psорт4(Grtr)}
    end),
    psорт4([Y || Y <- Xs, Y < X]) ++ [X] ++
    receive {Ref,Greater} -> Greater
end.
Benchmarks

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

```
5> foo:benchmark(psort4,L).
159.9
6> foo:benchmark(dsort,L).
182.13
```
An older slower machine... 2 cores no hyperthreading... silly to send it half the work

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

5> foo:benchmark(psort4,L).
159.9
6> foo:benchmark(dsорт,L).
182.13
7> foo:benchmark(dsорт2,L).
423.55
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*

```erlang
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

Client → Pool:
{get_node,ClientPid}
{use_node,Node}

Pool → Client:
{available,Node}
Node Pool Behaviour

\[
\text{pool([])} \rightarrow \\
\hspace{1cm} \text{receive} \\
\hspace{1cm} \{\text{available}, \text{Node}\} \rightarrow \\
\hspace{1cm} \text{pool([Node])} \\
\hspace{1cm} \text{end;} \\
\text{pool([Node|Nodes])} \rightarrow \\
\hspace{1cm} \text{receive} \\
\hspace{1cm} \{\text{get_node}, \text{Pid}\} \rightarrow \\
\hspace{1.5cm} \text{Pid} ! \{\text{use_node}, \text{Node}\}, \\
\hspace{2cm} \text{pool(Nodes)} \\
\hspace{1cm} \text{end.}
\]

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

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

Selective receive is really useful!
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.
Benchmarks

56> foo:benchmark(qsort,L).
321.01
57> foo:benchmark(psort4,L).
156.55
58> foo:benchmark(dsort2,L).
415.83
59> nodes().
[baz@HALL]
60> foo:benchmark(dwsort,L).
213.12
61> net_adm:ping('mary@CSE-360').
pong
62> nodes().
[baz@HALL,'mary@CSE-360']
63> foo:benchmark(dwsort,L).
269.71
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
