Erlang: An Overview

Part 5 – Parallel Programming in Erlang

Based on a lecture by John Hughes in his course on Parallel Functional Programming

Benchmarking programs

- Recall the Quick Sort function

```
qsort([]) -> [];
qsort([P|Xs]) ->
  qsort([X || X <- Xs, X =< P])
  ++ [P]  % pivot element
  ++ qsort([X || X <- Xs, P < X]).
```

- Let's create some test data for it

```
random_list(N) ->
  [rand:uniform(12345678) || _ <- lists:seq(1,N)].
```

```
4> L = qsort:random_list(200000).
... A random list with 200000 elements ...
5> timer:tc(qsort, qsort, [L]).
{427404,
 [42,237,342,401,593,623,858,911,959,1111,1144,1267,
  1402,1405,1529,1563,1638,1643,1729,1755,1864,1899,
  1926,1968,2014|...]}
```

microseconds

result

Benchmarking programs

- Let's define a benchmarking function

```
benchmark(Fun, L) ->
  Rs = [timer:tc(?MODULE, Fun, [L])
        || _ <- lists:seq(1, 100)],
  lists:sum([T || {T,_} <- Rs]) / (1000*length(Rs)).
```

- I.e. run 100 times, average and convert to msecs

```
10> qsort:benchmark(qsort, L).
427.64902
11> erlang:system_info(schedulers).
8
```

milliseconds

number of OS threads that the runtime system of the VM uses for running Erlang processes

Parallel sorting (naive)

- Let's parallelize the function (start of attempt)

sort elements greater than pivot in another process

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

how do we get the result here?

Parallel sorting (naive)

- Let’s parallelize the function (complete attempt)

get the Pid of the executing process

send the result back to the parent

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

wait to get the result of sorting the elements greater than pivot

```
14> qsort:benchmark(qsort, L).
427.64902
15> qsort:benchmark(pqsort, L).
826.27111
```

☹

Controlling granularity

```
pqsort2(L) -> pqsort2(5, L).

pqsort2(0, L) -> qsort(L);
pqsort2(_, []) -> [];
pqsort2(D, [P|Xs]) ->
  Par = self(),
  spawn_link(fun () ->
             Par ! pqsort2(D-1,[X || X <- Xs, P < X])
           end),
  pqsort2(D-1, [X || X <- Xs, X =< P])
  ++ [P]
  ++ receive Ys -> Ys end.
```

```
17> qsort:benchmark(qsort, L).
427.64902
18> qsort:benchmark(pqsort, L).
826.27111
19> qsort:benchmark(pqsort2,L).
236.19359
```

Correctness?

```
31> qsort:pqsort2(L) == qsort:qsort(L).
false
32> qsort:pqsort2("hello world").
" edhllloorw"
```

Oops!

What's going on?

```
pqsort2(D, [P|Xs]) ->
  Par = self(),
  spawn_link(fun () ->
               Par ! ...
             end),
  pqsort2(D-1, [X || X <- Xs, X =< P])
  ++ [P]
  ++ receive Ys -> Ys end.
```

What's going on?

```
pqsort2(D, [P|Xs]) ->
  Par1 = self(),
  spawn_link(fun () ->
                  Par1 ! ...
             end),
  Par = self(),
  spawn_link(fun () ->
                  Par ! ...
             end),
  pqsort2(D-2, [X || X <- Xs, X =< P])
  ++ [P]
  ++ receive Ys -> Ys end
  ++ [P1]
  ++ receive Ys1 -> Ys1 end.
```

Tagging messages

- Create a globally unique reference

```
Ref = make_ref()
```

- Send the message tagged with the reference

```
Par ! {Ref, Msg}
```

- Match the reference on receipt

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

- Picks the right message from the mailbox

A correct parallel sort

```
pqsort3(L) -> pqsort3(5, L).

pqsort3(0, L) -> qsort(L);
pqsort3(_, []) -> [];
pqsort3(D, [P|Xs]) ->
  Par = self(),
  Ref = make_ref(),
  spawn_link(fun () ->
               Gs = [X || X <- Xs, P < X],
               Par ! {Ref, pqsort3(D-1, Gs)}
             end),
  pqsort3(D-1, [X || X <- Xs, X =< P])
  ++ [P]
  ++ receive {Ref, Ys} -> Ys end.
```

Performance?

```
36> qsort:benchmark(qsort, L).
427.64902
37> qsort:benchmark(pqsort, L).
826.27111
38> qsort:benchmark(pqsort2, L).
236.19359
39> qsort:benchmark(pqsort3, L).
232.18068
```

What is copied here?

```
pqsort3(L) -> pqsort3(5, L).

pqsort3(0, L) -> qsort(L);
pqsort3(_, []) -> [];
pqsort3(D, [P|Xs]) ->
  Par = self(),
  Ref = make_ref(),
  spawn_link(fun () ->
                Gs = [X || X <- Xs, P < X],
                Par ! {Ref, pqsort3(D-1, Gs)}
              end),
  pqsort3(D-1, [X || X <- Xs, X =< P])
  ++ [P]
  ++ receive {Ref, Ys} -> Ys end.
```

terms in variables that the closure needs access to are copied to the heap of the spawned process

A parallel sort with less copying

```
pqsort4(L) -> pqsort4(5, L).

pqsort4(0, L) -> qsort(L);
pqsort4(_, []) -> [];
pqsort4(D, [P|Xs]) ->
  Par = self(),
  Ref = make_ref(),
  Gs = [X || X <- Xs, P < X],
  spawn_link(fun () ->
               Par ! {Ref, pqsort4(D-1, Gs)}
             end),
  pqsort4(D-1, [X || X <- Xs, X =< P])
  ++ [P]
  ++ receive {Ref, Ys} -> Ys end.
```

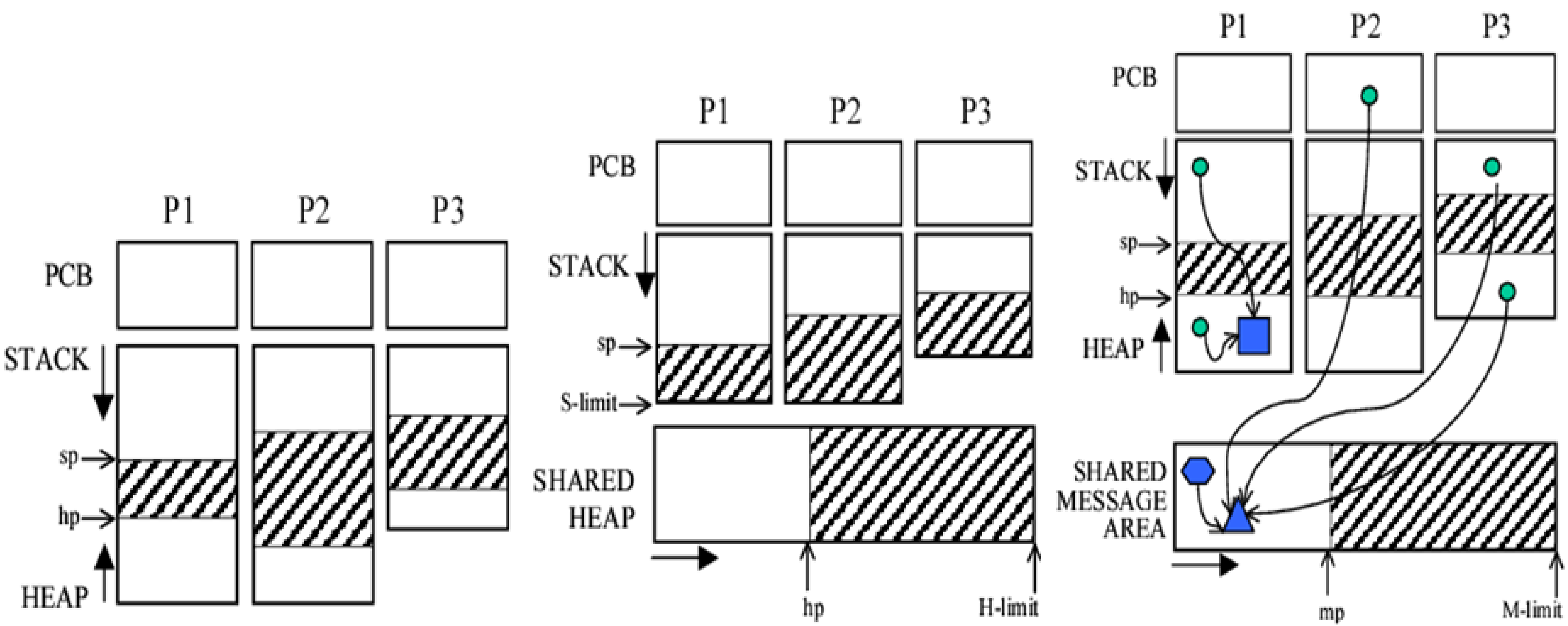
copy only the part of the list that the process needs to sort

Part 6 – A Glimpse of Erlang’s Implementation

Erlang's RunTime System (ERTS)

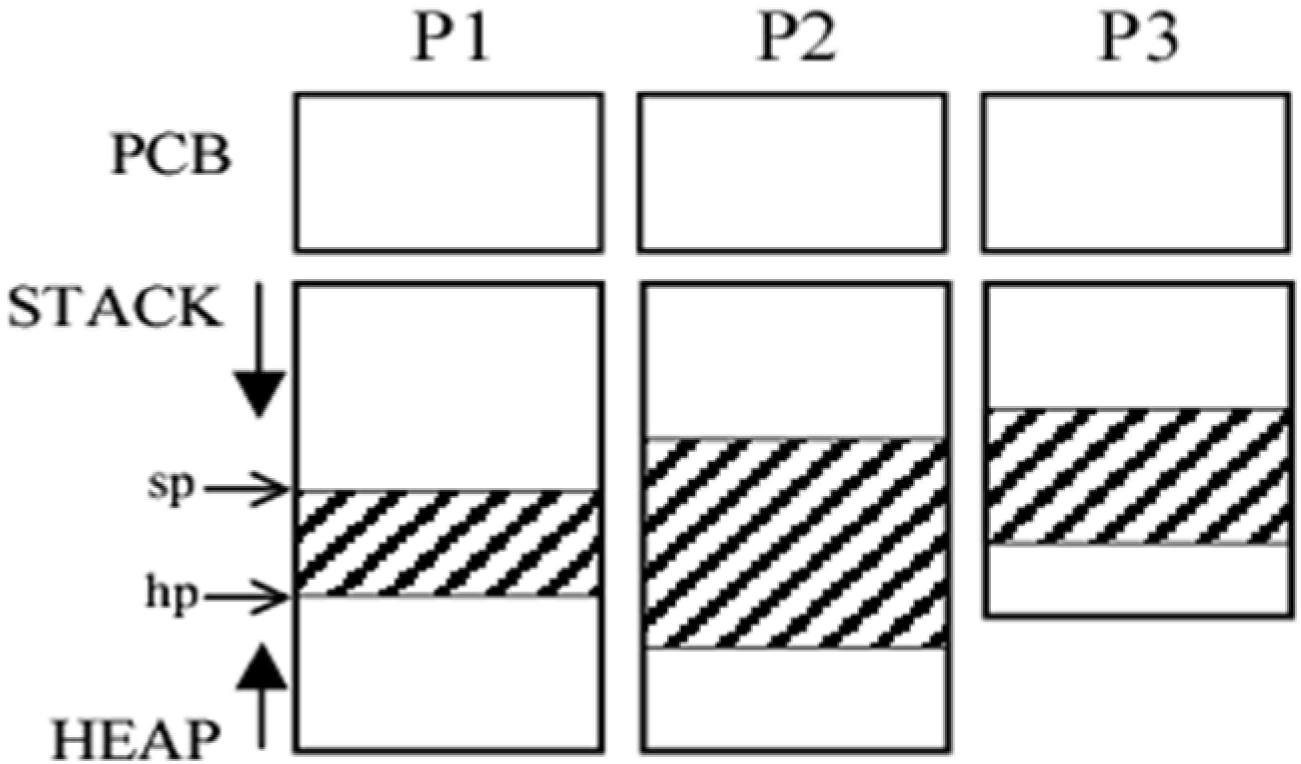
- Handles the basic "built-in" things:
  - memory allocation
  - garbage collection
  - process creation
  - message passing
  - context switching
- Several possible ways of structuring
- Some trade-offs have been studied
  - mainly on single core machines!

Runtime system architectures



(a) Process-centric (b) Communal (c) Hybrid architecture

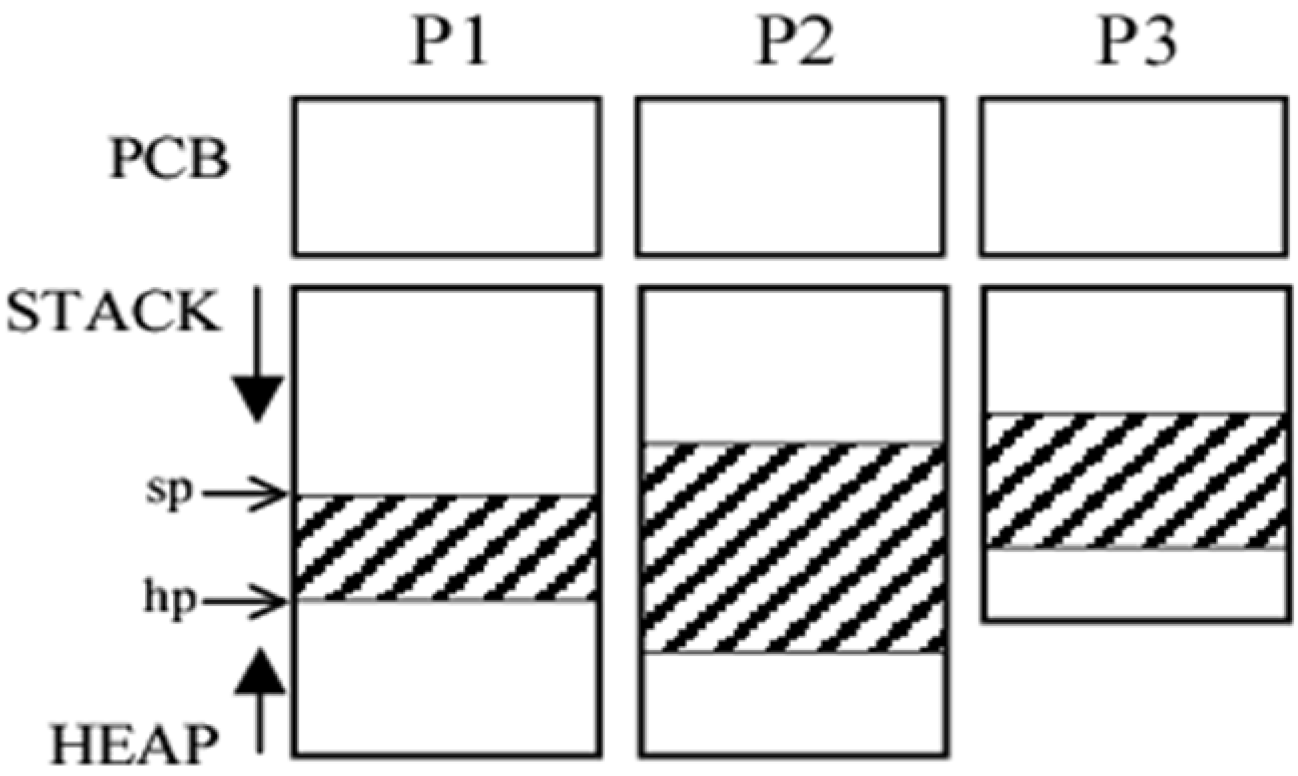
Process local heaps



Process local heaps

- Pros:
  - + Isolation and robustness
  - + Processes can be GC-ed independently
  - + Fast memory deallocation when a process terminates; processes used as regions/arenas
- Cons:
  - - Messages always copied, even between processes on the same machine
    - Sending is O(n) in the size of the message
  - - Memory fragmentation high

The truth...



Global areas:
- Atom table
- Process registry

Erlang Term Storage

“Big” Binary Area

ETS: Erlang Term Storage

- Key component of Erlang/OTP
  - Key/value store mechanism

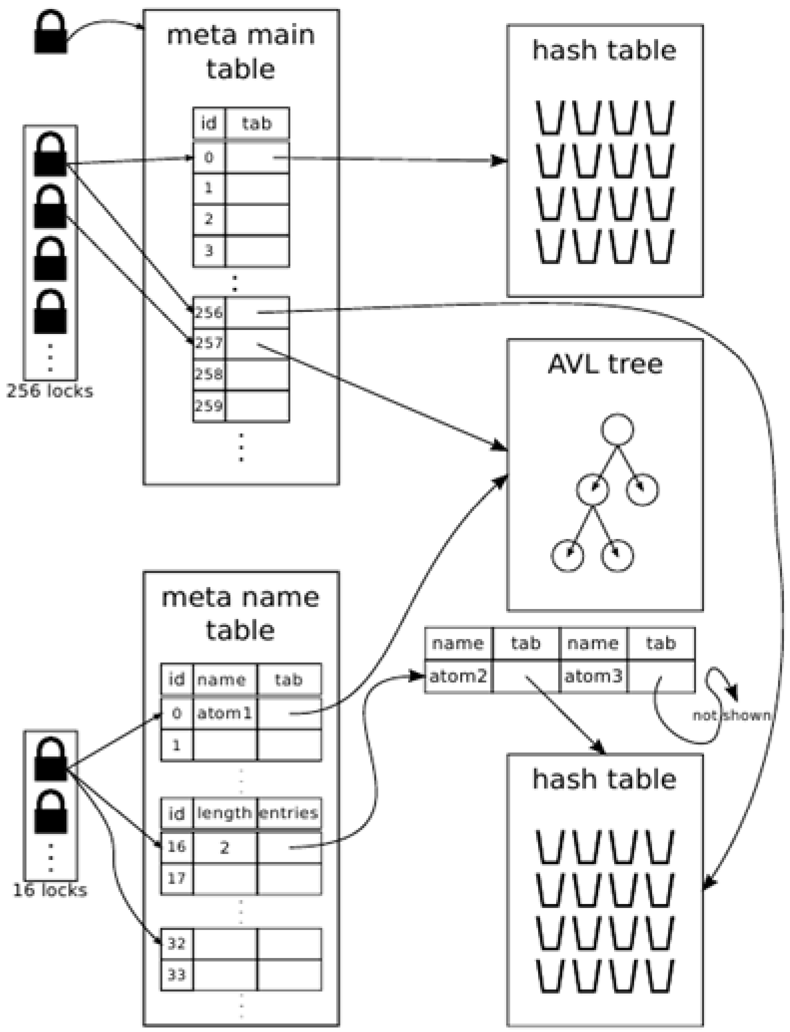
    in the form of tables that store tuples
  - Heavily used in applications
  - Supports the **`mnesia`** database
- Provides shared memory
  - with destructive updates!
  - sometimes crucial for parallelization

ETS example use

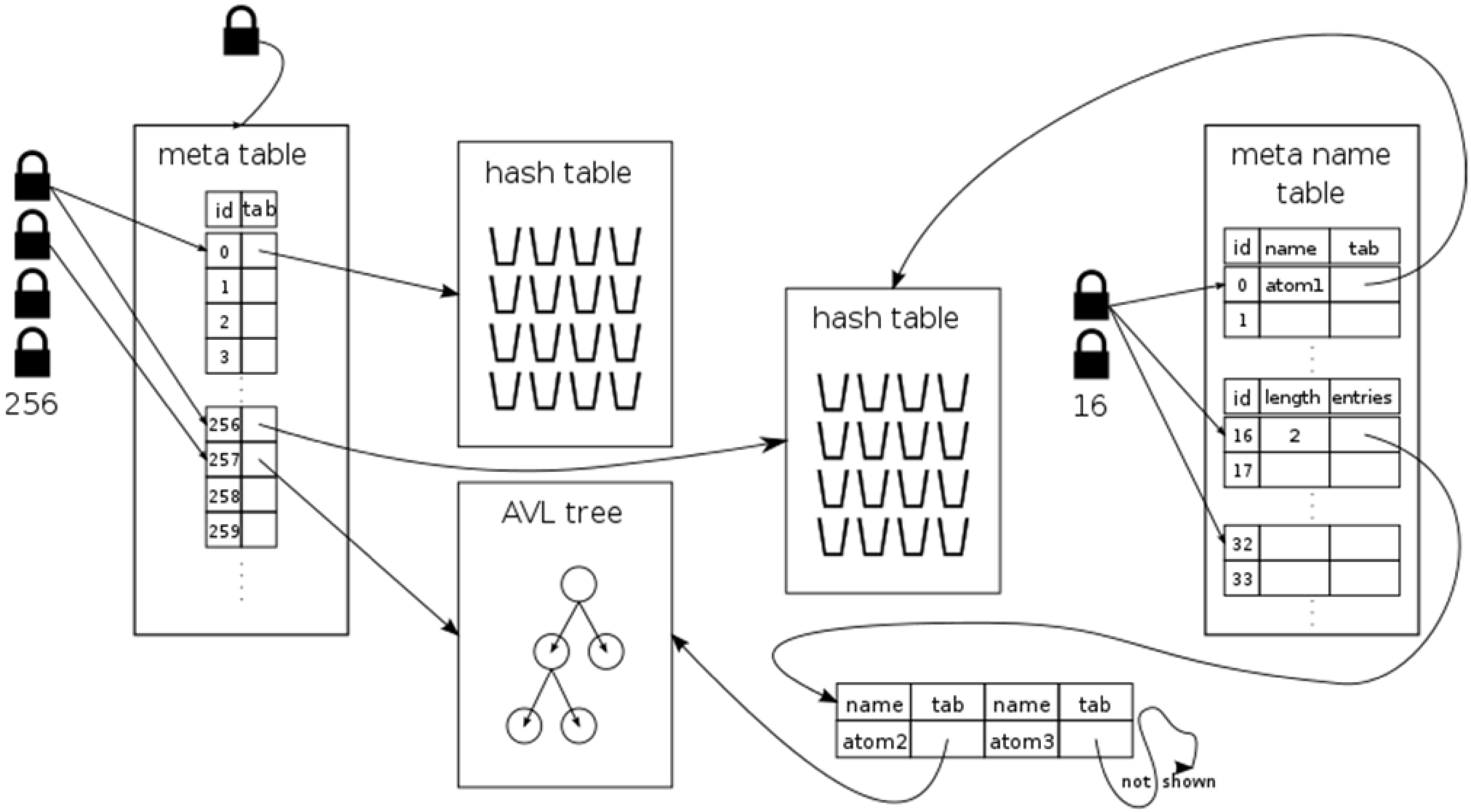
```
...
T = ets:new(mytable,
            [set, %bag, duplicate_bag, ordered_set
             public, %protected, private
             {keypos, 1},
             {read_concurrency, true},
             {write_concurrency, true}]),
ets:insert(T, [{key1,42}, {key2,val}]),
[{key1, V}] = ets:lookup(T, key1),
...
```

Implementation of ETS

- Four types/two implementations
  - **`set`**, **`bag`**, **`duplicate_bag`**
    - Linear Hash Tables
  - **`ordered_set`**
    - AVL Trees
- Concurrency options
  - **`write_concurrency`**
  - **`read_concurrency`**
  - reader groups (**`+rg`**)
    - fine-grained locks



ETS under the hood



Linear hash tables

- Hash key to bucket: bucket list
- Resizing one bucket at a time
  - Avg. bucket length: 6 in R16B

Locking

- One readers-writer table lock
- Bucket locks allow for fine-grained locking
- Some operations need to lock the whole table
  - Ex. insert all elements in a list atomically

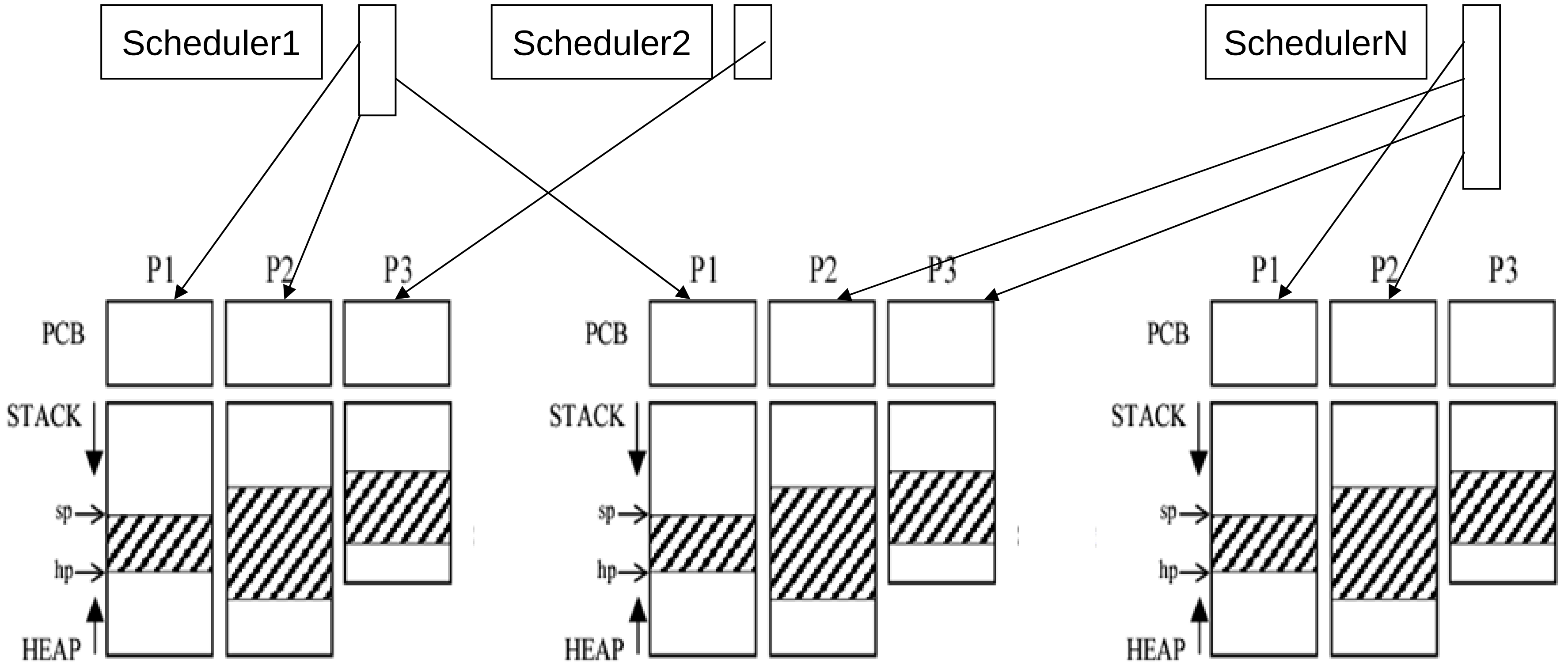
AVL trees

- Used for ETS tables of type **`ordered_set`**
- Balanced binary search trees

Locking

- Protected by *single* readers-writer lock

SMP architecture

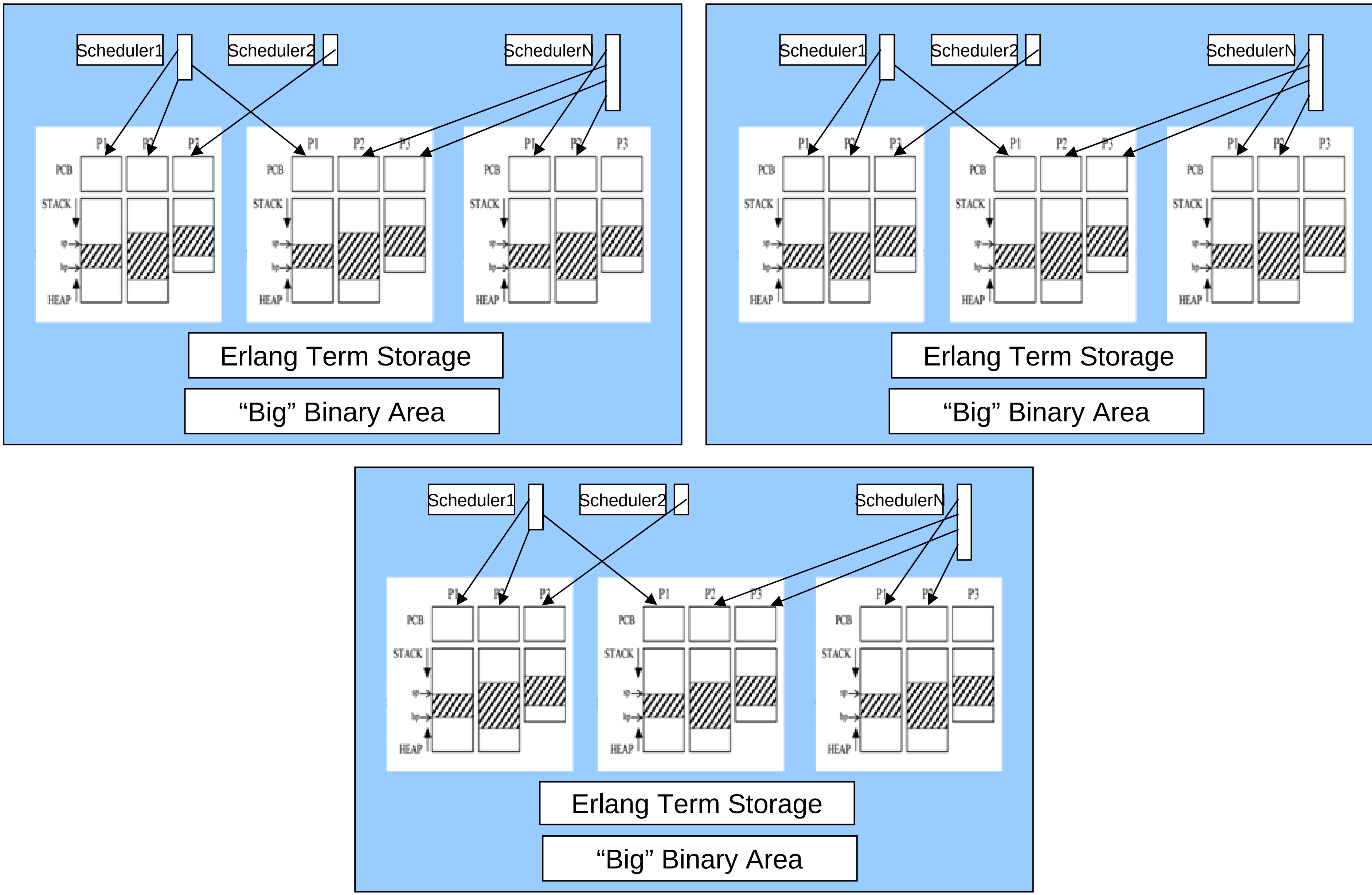


Global areas:
- Atom table
- Process registry

Erlang Term Storage

“Big” Binary Area

Distributed architecture



More information

Resources:
www.erlang.org

- Getting Started
- Erlang Reference Manual
- Library Documentation

Papers about Erlang and its implementation at:
http://www.it.uu.se/research/group/hipe

Information about Dialyzer at:
http://www.it.uu.se/research/group/hipe/dialyzer/
http://dialyzer.softlab.ntua.gr

References

1. E. Johansson, K. Sagonas, and J. Wilhelmsson. Heap Architectures for Concurrent Languages Using Message Passing. *ACM SIGPLAN Notices*, 38(2):88-99, Feb. 2002. ACM Press. doi: 10.1145/773039.512440

2. R. Carlsson, K. Sagonas, and J. Wilhelmsson. Message Analysis for Concurrent Programs that use Message Passing. *ACM TOPLAS*, 28(4):715-746, July 2006. doi: 10.1145/1146809.1146813

3. S. Aronis and K. Sagonas. On using Erlang for parallelization: Experience from parallelizing Dialyzer. In *Trends in Functional Programming*, pages 295-310, Vol. 7829 in LNCS, June 2012. Springer. doi: 10.1007/978-3-642-40447-4_19

4. S. Aronis *et al*. A scalability benchmark suite for Erlang/OTP. In *Proceedings of the 11th International Workshop on Erlang*, pages 33-42, Sept. 2012. ACM Press. doi: 10.1145/2364489.2364495

5. D. Klaftenegger, K. Sagonas, and K. Winblad. On the Scalability of the Erlang Term Storage. In *Proceedings of the 12th International Workshop on Erlang*, pages 15-26, Sept. 2013. ACM Press. doi: 10.1145/2505305.2505308

6. K. Sagonas, and K. Winblad. More Scalable Ordered Set for ETS Using Adaptation. In *Proceedings of the 13th International Workshop on Erlang*, pages 3-11, Sept. 2014. ACM Press. doi: 10.1145/2633448.2633455

7. P. Trinder, N. Chechina, N. Papaspyrou, K. Sagonas, S. Thompson, *et al*. Scaling Reliably: Improving the Scalability of the Erlang Distributed Actor Platform. *ACM TOPLAS*, 39(4), 17, Sept. 2017. doi: 10.1145/3107937