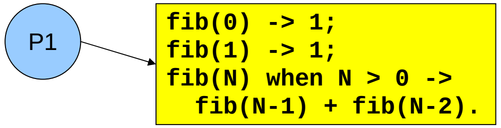
Erlang: An Overview

Part 2 – Concurrency and Distribution

Thanks to Richard Carlsson for most of the slides in this part

---

Processes



```
fib(0) -> 1;
fib(1) -> 1;
fib(N) when N > 0 ->
  fib(N-1) + fib(N-2).
```

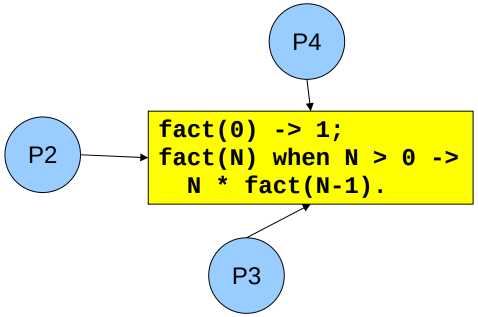
- Whenever an Erlang program is running, the code is executed by a *process*
- The process keeps track of the current program point, the values of variables, the call stack, etc.
- Each process has a unique *Process Identifier (“Pid”),* that can be used to identify the process
- *Processes are concurrent* (they can run in parallel)

Implementation

- Erlang processes are implemented by the VM’s runtime system, not by operating system threads
- Multitasking is *preemptive* (the virtual machine does its own process switching and scheduling)
- Processes use very little memory, and switching between processes is very fast
- Erlang VM can handle large numbers of processes
  - Some applications use more than 100.000 processes
- On a multiprocessor/multicore machine, Erlang processes can be scheduled to run in parallel on separate CPUs/cores using multiple schedulers

---

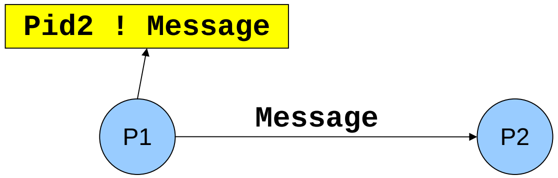
Concurrent process execution



```
fact(0) -> 1;
fact(N) when N > 0 ->
  N * fact(N-1).
```

- Different processes may be reading the same program code at the same time
  - They have their own data, program point, and stack – only the text of the program is being shared (well, almost)
  - *The programmer does not have to think about other processes updating the variables*

Message passing



```
Pid2 ! Message
```

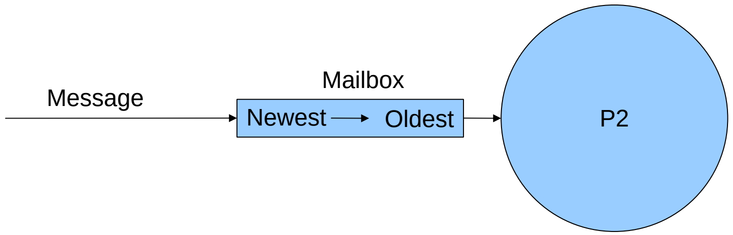
P1

Message

P2

- “`!`” is the *send operator* (often called “bang!”)
  - The Pid of the receiver is used as the address
- Messages are sent *asynchronously*
  - The sender continues immediately
- Any value can be sent as a message

Message queues



Message

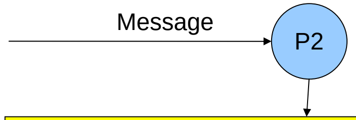
Mailbox

Newest → Oldest

P2

- Each process has a *message queue* (mailbox)
  - Arriving messages are placed in the queue
  - *No size limit* – messages are kept until extracted
- A process *receives* a message when it extracts it from the mailbox
  - Does not have to take the first message in the queue

Receiving a message



Message

P2

```
receive
  Msg -> io:format("~w\n", [Msg])
end
```

`receive` expressions are similar to **`case`** switches

- Patterns are used to match messages in the mailbox
- Messages in the queue are tested in order
  - The first message that matches will be extracted
  - A variable-pattern will match the first message in the queue
- Only one message can be extracted each time

Selective receive

```
receive
  {foo, X, Y} -> ...;
  {bar, X} when ... -> ...;
  ...
end
```

- Patterns and guards let a programmer control the priority with which messages will be handled
  - Any other messages will remain in the mailbox
- The **`receive`** clauses are tried in order
  - If no clause matches, the next message is tried
- If *no* message in the mailbox matches, the process *suspends*, waiting for a new message

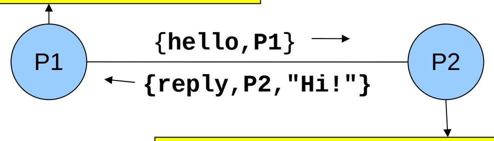
Receive with timeout

```
receive
   {foo, X, Y} -> ...;
   {bar, X} when ... -> ...
after 1000 ->
   ...        % handle timeout
end
```

- A **`receive`** expression can have an **`after`** part
  - The timeout value is either an integer (milliseconds), or the atom **`'infinity'`** (wait forever)
  - 0 (zero) means “just check the mailbox, then continue”
- The process will wait until a matching message arrives, or the timeout limit is exceeded
- **Soft real-time**: approximate, no strict timing guarantees

Send and reply

```
Pid ! {hello, self()},
receive
   {reply, Pid, String} ->
      io:put_chars(String)
end
```

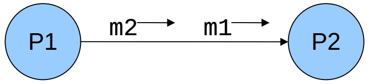


```
receive
   {hello, Sender} ->
      Sender ! {reply, self(), "Hi!"}
end
```

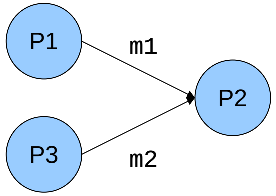
- Pids are often included in messages (**`self()`**), so the receiver can reply to the sender
  - If the reply includes the **`Pid`** of the second process, it is easier for the first process to recognize the reply

Message order

FIFO order
(same pair of sender and receiver)

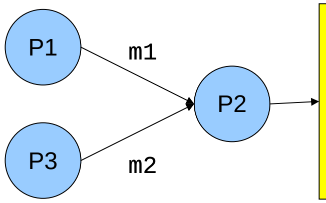


No guaranteed order
(different senders, same receiver)



- Within a node, the only guaranteed message order is when both the sender and receiver are the same for both messages (First-In, First-Out)
  - In the left figure, m1 will always arrive before m2 in the message queue of P2 (if m1 is sent before m2)
  - In the right figure, the arrival order can vary

Selecting unordered messages



```
receive
   m1 -> io:format("Got m1!")
end,
receive
   m2 -> io:format("Got m2!")
end
```

- Using selective receive, we can choose which messages to accept, even if they arrive in a different order
- In this example, P2 will always print “Got m1!” before “Got m2!”, even if m2 arrives before m1
  - m2 will be ignored until m1 has been received

Starting processes

- The `'spawn'` function creates a new process
- There are several versions of '`spawn`':
  - `spawn(fun() -> ... end)`
    - can also do `spawn(fun f/0)` or `spawn(fun m:f/0)`
  - `spawn(Module, Function, [Arg1, ..., ArgN])`
    - `Module:Function/N` must be an exported function
- The new process will run the specified function
- The spawn operation always returns immediately
  - The return value is the Pid of the new process
  - The “parent” always knows the Pid of the “child”
  - The child will not know its parent unless you tell it

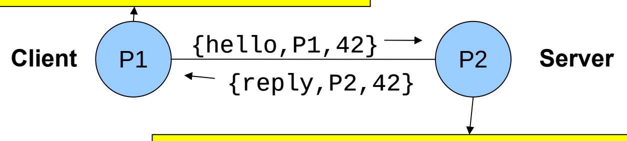
Process termination

- A process *terminates* when:
  - It finishes the function call that it started with
  - There is an exception that is not caught
    - The purpose of '`exit`' exceptions is to terminate a process
    - “`exit(normal)`” is equivalent to finishing the initial call
- All messages sent to a terminated process will be thrown away, without any warning
  - No difference between throwing away and putting in mailbox just before process terminates
- The same process identifier will not be used again for a long time

---

A stateless server process

```
run() ->
  Pid = spawn(fun echo/0),

  Pid ! {hello, self(), 42},
  receive
    {reply, Pid, 42} ->
      Pid ! stop
  end.
```



Client

P1

{hello,P1,42}

P2

Server

{reply,P2,42}

```
echo() ->
  receive
    {hello, Sender, Value} ->
      Sender ! {reply, self(), Value},
      echo();  % loop!
    stop ->
      ok
  end.
```

A server process with state

```
server(State) ->
  receive
    {get, Sender} ->
      Sender ! {reply, self(), State},
      server(State);
    {set, Sender, Value} ->
      Sender ! {reply, self(), ok},
      server(Value);    % loop with new state!
    stop ->
      ok
  end.
```

- The parameter variables of a server loop can be used to remember the current state
- Note: the recursive calls to **`server()`** are *tail calls (last calls) – the loop does not use stack space*
- *A server like this can run forever*

A simple server example

```
-module(simple_server).
-export([start/0]).

-spec start() -> pid().
start() ->
  spawn(fun() -> loop(0) end).

-spec loop(integer()) -> no_return().
loop(Count) ->
  NC = receive
         {report, Pid} -> Pid ! Count;
         _AnyOtherMsg  -> Count + 1
       end,
  loop(NC).
```

```
Eshell V5.10.3 (abort ...^G)
1> P = simple_server:start().
<0.42.0>
2> P ! foo.
foo
3> [P ! X || lists:seq(1,9)].
[1,2,3,4,5,6,7,8,9]
4> P ! {report, self()},
   receive M -> M end.
10
```

Hot code swapping

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

start() -> spawn(fun() -> loop(0) end).

loop(State) ->
  receive
    {get, Sender} ->
      ...,
      server:loop(State);
    {set, Sender, Value} ->
      ...,
      server:loop(Value);
  ...
```

- When we use "**`module:function(...)`**", Erlang will always call the latest version of the module
  - If we recompile and reload the **`server`** module, the process will jump to the new code after handling the next message – we can fix bugs without restarting!

Hiding message details

```
get_request(ServerPid) ->
  ServerPid ! {get, self()}.

set_request(Value, ServerPid) ->
  ServerPid ! {set, self(), Value}.

wait_for_reply(ServerPid) ->
  receive
    {reply, ServerPid, Value} -> Value
  end.

stop_server(ServerPid) ->
  ServerPid ! stop.
```

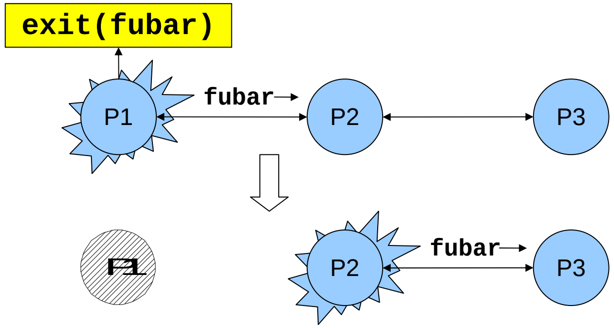
- Using interface functions keeps the clients from knowing about the format of the messages
  - You may need to change the message format later
- It is the client who calls the **`self()`** function here

Registered processes

```
Pid = spawn(...),

register(my_server, Pid),

my_server ! {set, self(), 42},

42 = get_request(my_server),

Pid = whereis(my_server)
```

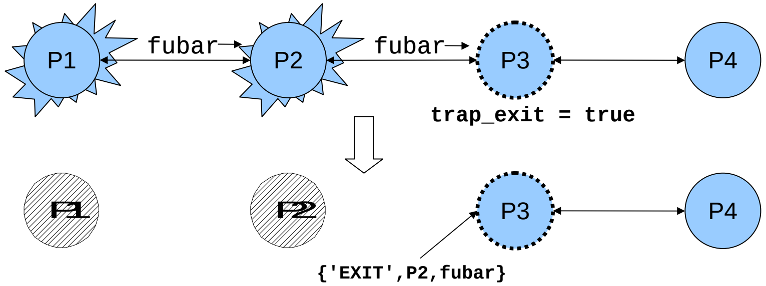
- A process can be registered under a name
  - the name can be any atom
- Any process can send a message to a registered process, or look up the Pid
- The Pid might change (if the process is restarted and re-registered), but the name stays the same

Links and exit signals



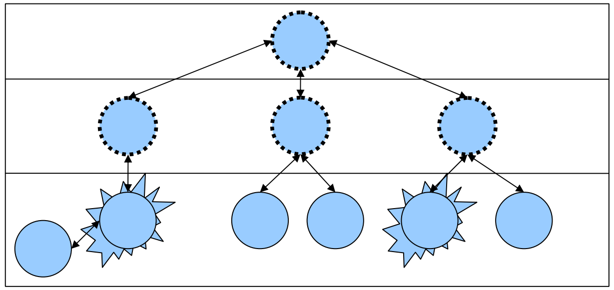
- Any two processes can be *linked*
  - Links are always bidirectional (two-way)
- When a process dies, an *exit signal* is sent to all linked processes, which are also killed
  - Normal exit does not kill other processes

Trapping exit signals



- If a process sets its **`trap_exit`** flag, all signals will be caught and turned into normal messages
  - **`process_flag(trap_exit, true)`**
  - **`{'EXIT', Pid, ErrorTerm}`**
- This way, a process can watch other processes
  - 2-way links guarantee that sub-processes are dead

Robust systems through layers



- Each layer supervises the next layer and restarts the processes if they crash
- The top layers use well-tested, very reliable libraries (OTP) that practically never crash
- The bottom layers may be complicated and less reliable programs that can crash or hang

---

Distribution

```
[foo.bar.se] $ erl -name fred
Erlang (BEAM) emulator version 5.10.3

Eshell V5.10.3 (abort with ^G)
(fred@foo.bar.se)1> node().
'fred@foo.bar.se'
(fred@foo.bar.se)2>
```

- Running “**`erl`**” with the flag “**`-name xxx`**”
  - starts the Erlang network distribution system
  - makes the virtual machine emulator a “*node*”
    - the node name is the atom '**`xxx@host.domain`**'
- Erlang nodes can communicate over the network
  - but first they must find each other
  - simple security based on secret cookies

Connecting nodes

```
(fred@foo.bar.se)2> net_adm:ping('barney@foo.bar.se').
pong
(fred@foo.bar.se)3> net_adm:ping('wilma@foo.bar.se').
pang
(fred@foo.bar.se)4>
```

- Nodes are connected the first time they try to communicate – after that, they stay in touch
  - A node can also supervise another node
- The function “**`net_adm:ping(Node)`**” is the easiest way to set up a connection between nodes
  - returns either “**`pong`**” or “**`pang`**” ☺
- We can also send a message to a registered process using “**`{Name,Node} ! Message`**”

Distribution is transparent

- One can send a Pid from one node to another
  - Pids are unique, even over different nodes
- We can send a message to *any* process through its Pid – even if the process is on another node
  - There is no difference (except that it takes more time to send messages over networks)
  - We don't have to know where processes are
  - We can make programs work on multiple computers with no changes at all in the code (no shared data)
- We can run several Erlang nodes (with different names) on the same computer – good for testing

---

Running remote processes

```
P = spawn('barney@foo.bar.se', Module, Function, ArgList),

global:register_name(my_global_server, P),

global:send(my_global_server, Message)
```

- We can use variants of the **`spawn`** function to start new processes directly on another node
- The module **`'global'`** contains functions for
  - registering and using named processes over the whole network of connected nodes
    - not same namespace as the local “**`register(...)`**”
    - must use “**`global:send(...)`**”, not “`!`”
  - setting global locks

Ports – talking to the outside

```
PortId = open_port({spawn, "command"}, [binary]),

PortId ! {self(), {command, Data}}

PortId ! {self(), close}
```

- Talks to an external (or linked-in) C program
- A port is connected to the process that opened it
- The port sends data to the process in messages
  - binary object
  - packet (list of bytes)
  - one line at a time (list of bytes/characters)
- A process can send data to the port

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) ->
  [random: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
```

☺

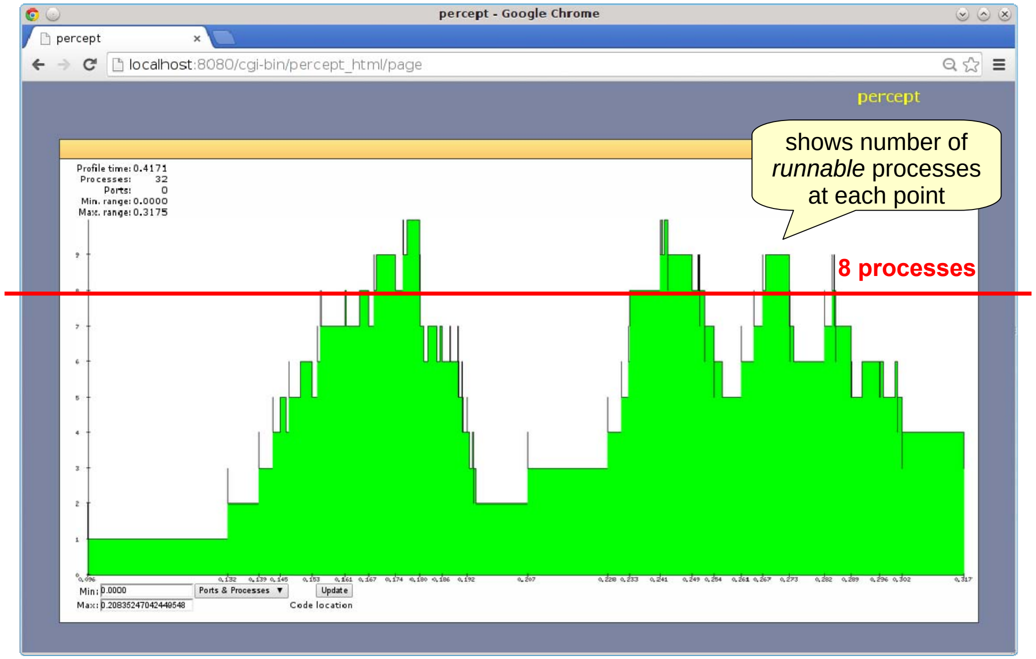
Profiling parallelism

file to store profiling information in

function to profile

```
21> percept:profile("prof.data",
                    {qsort,pqsort2,[L]}, [procs]).
Starting profiling.
ok
22> percept:analyze("prof.data").
Parsing: "prof.data"
Consolidating...
Parsed 255 entries in 0.116107 s.
    32 created processes.
     0 opened ports.
ok
23> percept:start_webserver(8080).
{started,"laptop",8080}
```

Profiling with percept



shows number of *runnable* processes at each point

8 processes

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

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

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