Concurrency, Parallelism and Distribution (CPD)

Concurrency: Lists, Parallelism and Dynamic Code Loading in Erlang

Joaquim Gabarro
(gabarro@cs.upc.edu)

Computer Science
BarcelonaTech, Universitat Politècnica de Catalunya

Basic reading

(1) Joe Armstrong
Programming Erlang, Software for a Concurrent World
Pragmatic Bookshelf, 2007.

Download the code from:

`https://pragprog.com/book/jaerlang/programming-erlang`

(2) Joe Armstrong, Erlang
CACM, September 2010, Vol 53, No 9, 68-75

(3) Jim Larson, Erlang for Concurrent Programming
CACM, March 2009, Vol 52, No 3, 48-56

Slides are based on Armstrong's book

Lists

Execution time & Straightforward parallelism

More on messages and processes

Dynamic code loading

1. Lists

Lists

- We use lists to store variable numbers of things.
- We create a list by enclosing the list elements in square brackets and separating them with commas.

Example: Shopping list:

```
1> ThingsToBuy = [{apples,10},{pears,6},{milk,3}].
[{apples,10},{pears,6},{milk,3}]
```

- The first element of a list is the head of the list.
- If you remove the head, what's left is the tail of the list.

Defining Lists (1)

- If T is a list, [H|T] is also a list, with head H and tail T.
- The vertical bar | separates the head from its tail.
- [ ] is the empty list.

```
7> [a| [b,c]].
[a,b,c]
8> [[2]|[b,c]].
[[2],b,c]
9> [2|b,c].
* 1: syntax error before: ','
9> [a|[]].
[a]
```

Defining Lists (2)

We can add more than one element to the beginning of T by writing [E1,E2,..,En|T].

```
3> ThingsToBuy1 = [{oranges,4},{newspaper,1}|ThingsToBuy].
[{oranges,4},{newspaper,1},{apples,10},{pears,6},{milk,3}]
```

Extracting Elements

If we have a nonempty list a L, then the expression [X|Y] = L, where X and Y are unbound variables, will extract the head of the list into X and the tail of the list into Y.

Example: We have our shopping list ThingsToBuy1

```
13> [Buy1|ThingsToBuy2] = ThingsToBuy1.
[{oranges,4},{newspaper,1},{apples,10},{pears,6},{milk,3}]
14> Buy1.
{oranges,4}
15> ThingsToBuy2.
[{newspaper,1},{apples,10},{pears,6},{milk,3}]
```

Simple list processing (mylists.erl)

sum(L) computes the sum of the elements in L

```
sum([H|T]) -> H + sum(T);
sum([]) -> 0.
```

Execution example:

```
16> cd("D:/.../Erlang_code").
..
17> c(mylists)
...
22> L=[1,2,3].
[1,2,3]
23> mylists:sum(L).
6
```

Common list processing functions (mylists.erl)

map(F, L) returns a list where every element is obtained applying F to the corresponding element in L.

```
map(_, [])       -> [];
map(F,  [H|T]) -> [F(H)|map(F, T)].
```

member(X, L) returns true if X is an element of L, false otherwise.

```
member(H, [H|_]) -> true;
member(H, [_|T]) -> member(H, T);
member(_, [])    -> false.
```

reverse(L) reverses the order of the elements in L

```
reverse(L) -> reverse(L, []).
reverse([H|T], L) -> reverse(T, [H|L]);
reverse([], L)    -> L.
```

Strings and Lists (1)

When the shell prints the value of a list it prints the list as a string, but only if all the integers represent printable values.

```
1> L1= [1,2,3,83,117,114,112,114,105,115,101].
...
2> L1.
[1,2,3,83,117,114,112,114,105,115,101]
3> L2= [83,117,114,112,114,105,115,101].
...
6> L2.
"Surprise"
```

American Standard Code for Information Interchange, ASCII:

$$\mathtt{83} \rightarrow \mathtt{S}, \mathtt{117} \rightarrow \mathtt{u}, \ldots, \mathtt{101} \rightarrow \mathtt{3}$$

Strings and Lists (2)

sep(L, N) returns {L1,L2} so that L1++L2 == L, length(L1)=N .

```
sep(L, 0) -> {[], L};
sep([H|T], N) ->
     {L1, L2} = sep(T, N-1),
     {[H|L1], L2}.
```

The call `sep(L,3)` returns `{[1,2,3],"Surprise"}`

List comprehension

- Is construct for creating a list based on existing lists.
- It follows the form of the mathematical set-builder notation in the Zermelo-Frankel theory.

Example: set-builder notation:

$$S = \{\, 2 \cdot x \mid x \in \{0, \ldots, 100\},\ x^2 > 3,\ x^2 < 50 \}$$

$$S = \{ \underbrace{2 \cdot x}_{\text{output function}} \mid \underbrace{x}_{\text{variable}} \in \underbrace{\{0, \ldots, 100\}}_{\text{input set}}, \underbrace{x^2 > 3}_{\text{predicate}}, \underbrace{x^2 < 50}_{\text{predicate}} \}$$

Erlang translation

```
24> S = [2*X || X <- lists:seq(0,100), X*X > 3, X*X<50].
[4,6,8,10,12,14]
```

List comprehension, examples

```
27> [{X,Y}|| X<-[1,2,3], Y<-[a,b,c]].
[{1,a},{1,b},{1,c},{2,a},{2,b},{2,c},{3,a},{3,b},{3,c}]
28> [X|| X<-[1,2,3,4], Y<-[3,4,5,6], X==Y].
[3,4]
29> [X+1|| X<-[1,2,3,4], X rem 2==0].
[3,5]
```

General form `[Exp || Gen1,...,GenN, Filter1,...,FilterN]`
Example: permutations

```
-module(perm).
-compile([export_all]).

perm([])->[[]];
perm(L)->[[H|T] || H<-L, T<-perm(L--[H])].
```

Execution example:

```
45> perm:perm([c,a,t]).
[[c,a,t],[c,t,a],[a,c,t],[a,t,c],[t,c,a],[t,a,c]]
```

Sequential quicksort

```
-module(pqs).
-compile([export_all]).

%% sequential quicksort
qs([]) -> [];
qs([H|T]) ->
    LT = [X || X <- T, X < H],
    GE = [X || X <- T, X >= H],
    qs(LT) ++ [H] ++ qs(GE).
```

Example: Given $[27, 82, 43, 15, 10, 38, 9, 3]$

```
[H|T]=[27|82,43,15,10,38,9,3]
LT=[15,10,9,3], GE=[82,43,38]
qs(LT)=[3,9,10,15], qs(GE)=[38,43,82]
qs(LT)++[H]++qs(GE)=[3,9,10,15,27,38,43,82]
```

2. Execution time & Straightforward parallelism

Execution time

apply(M, F, [Arg1,...,ArgN]) equivalent M:F(Arg1,...,ArgN)

chrono(M, F, P) returns the compuing time of M:F having P as parameters.

```
-module(new_pqs).
-compile([export_all]).
....
chrono(M, F, P) ->
    {_, Seconds, Micros} = erlang:timestamp(),
    T1 = Seconds + (Micros/1000000.0),
    apply(M, F, P),
    {_, Seconds2, Micros2} = erlang:timestamp(),
    T2 = Seconds2 + (Micros2/1000000.0),
    T2 - T1.
```

qs execution time

Macro ?MODULE expands to the current module name.

```
-module(pqs).
-compile([export_all]).

%% test functions
random_list(N) -> random_list(N, N, []).
random_list(0, _, L) -> L;
random_list(N, M, L)
   -> random_list(N-1, M, [random:uniform(M) | L]).

test_seq(N)->L=random_list(N),chrono(?MODULE, qs, [L]).
```

Execute a test

```
2> c(new_pqs).
{ok,new_pqs}
3> new_pqs:test_seq(5000000).
9.953000000037719
```

Parallel quicksort pqs

```
...
%% parallel quicksort
pqs(L) -> P = spawn(?MODULE, pqs2, [self(), L]), rcv(P).

rcv(P) ->receive {P, X} -> X end.

pqs2(P, L) ->
    if
        length(L) < 100000 ->
            P ! {self(), qs(L)};
        true ->
            [H | T] = L,
            LT = [X || X <- T, X < H],
            GE = [X || X <- T, X >= H],
            P1 = spawn(?MODULE, pqs2, [self(), LT]),
            P2 = spawn(?MODULE, pqs2, [self(), GE]),
            L1 = rcv(P1),
            L2 = rcv(P2),
            P ! {self(), L1 ++ [H] ++ L2}
    end.
```

qs versus pqs

Explicit parallelism does not help very much:

```
2> c(new_pqs).
{ok,new_pqs}
3> new_pqs:test_seq(5000000).
9.953000000037719
4> new_pqs:test_par(5000000).
8.172000000020489
```

4. More on messages and processes

Receive with a Timeout

A receive statement might wait forever for a message that never comes.

```
receive
    Pattern1 [when Guard1] -> Expressions1;
    Pattern2 [when Guard2] ->Expressions2;
    ...
after Time ->
     Expressions
end.
```

If no matching message has arrived within Time milliseconds of entering the receive expression, then the process will stop waiting for a message and evaluate Expressions.

Example : Sleep

Function sleep(T) suspends the current process for T milliseconds.

Download `lib_misc.erl`

```
sleep(T) ->
    receive
    after T ->
        true
    end.
```

Example: Receive with Timeout Value of Zero (1)

The function flush_buffer entirely empties the mailbox of a process.

Download `lib_misc.erl`

```
flush_buffer() ->
    receive
        _Any ->flush_buffer()
    after 0 ->
        true
    end.
```

Without the timeout clause, flush_buffer would suspend forever and not return when the mailbox was empty.

Example: Receive with Timeout Value of Zero (2)

We use a zero timeout to implement a form of priority receive as follows (download `lib_misc.erl`)

```
priority_receive() ->
    receive
        {alarm, X} -> {alarm, X}
    after 0 ->
        receive
            Any -> Any
         end
    end.
```

- If there is no message at all, it will suspend in the innermost receive and return the first message it receives.
- If there is a message matching alarm, X, then this message will be returned immediately.
- Remember that the after section is checked only after pattern matching has been performed on all the entries.

Example: Implementing a Timer (1)

- The function stimer:start(Time, Fun) will evaluates Fun after Time ms.
- It returns a handle (which is a PID) used to cancel the timer if required.

(download `stimer.erl`)

```
-module(stimer).
-export([start/2, cancel/1]).

start(Time, Fun)->spawn(fun()->timer(Time, Fun) end).
cancel(Pid) -> Pid ! cancel.

timer(Time, Fun) ->
    receive
        cancel ->void
    after Time ->
         Fun()
    end.
```

Example: Implementing a Timer (2)

We can test this as follows:

I wait more than five seconds so that the timer would trigger.

```
1> Pid = stimer:start(5000, fun() ->
                      io:format("timer event~n") end).
<0.42.0>
timer event
```

I'll start a timer and cancel it before has expired:

```
2> Pid1 = stimer:start(25000, fun()
                      -> io:format("timer event~n") end).
<0.49.0>
3> stimer:cancel(Pid1).
cancel
```

Selective Receive (1)

- Each process in Erlang has an associated mailbox.
- When you send a message to the process, the message is put into the mailbox.
- The only time the mailbox is examined is when your program evaluates a receive. statement.

```
receive
    Pattern1 [when Guard1] -> Expressions1;
    Pattern2 [when Guard1] -> Expressions1;
...
after
    Time -> ExpressionTimeout
end
```

Selective Receive (2)

Receive works as follows:

- When we enter a receive statement, we start a timer.
- Take the first message in the mailbox and try to match it against Pattern1, Pattern2, and so on. If the match succeeds, the message is removed from the mailbox, and the expressions following the pattern are evaluated.
- If none of the patterns in the receive statement matches the first message in the mailbox, then the first message is removed from the mailbox and put into a save queue. The second message in the mailbox is then tried. This procedure is repeated until a matching message is found or until all the messages in the mailbox have been examined.

Selective Receive (3)

- If none of the messages in the mailbox matches, then the process is suspended and will be rescheduled for execution the next time a new message is put in the mailbox. Note that when a new message arrives, the messages in the save queue are not rematched.
- As soon as a message has been matched, then all messages that have been put into the save queue are reentered into the mailbox in the order in which they arrived at the process. If a timer was set, it is cleared.
- If the timer elapses when we are waiting for a message, then evaluate the expressions ExpressionsTimeout and put any saved messages back into the mailbox in the order in which they arrived at the process.

Registered Processes

Erlang has a method for publishing a process identifier so that any process in the system can communicate with this process.

register(AnAtom, Pid). Register the process Pid with the name AnAtom. The registration fails ifAnAtom has already been used to register a process.

```
1> Pid = spawn(fun area_server0:loop/0).
<0.51.0>
2> register(area, Pid).
true
3> area ! {rectangle, 4, 5}.
Area of rectangle is 20
{rectangle,4,5}
```

A Clock (1)

We use register to make a registered process that represents a clock (download `clock.erl`).

```
-module(clock).
-export([start/2, stop/0]).

start(Time, Fun) ->
    register(clock, spawn(fun() -> tick(Time, Fun) end)).

stop() -> clock ! stop.

tick(Time, Fun) ->
    receive
        stop -> void
    after Time ->
        Fun(), tick(Time, Fun)
    end.
```

A Clock (2)

The clock will happily tick away until you stop it:

```
3> clock:start(5000, fun()
          -> io:format("TICK ~p~n",[erlang:now()]) end).
true
TICK {1350,553188,828000}
TICK {1350,553193,843000}
TICK {1350,553198,859000}
TICK {1350,553203,874000}
4> clock:stop().
stop
```

5. Dynamic code loading

Dynamic code loading: simple idea

Dynamic code loading is one surprising features of Erlang.

The idea is simple: every time we call

someModule:someFunction(...)

we'll always call the latest version of the function in the latest version of the module, even if we recompile the module while code is running in this module.

Example: a and b (1)

Consider two little modules a and b.

- If a calls b in a loop and we recompile b, then a will automatically call the new version of b the next time b is called.
- If different processes a are running and all of them call b, then all of them will call the new version of b if b is recompiled.

Module b is very simple

```
-module(b).
-export([x/0]).

x() -> 1.
```

Example: Module a (2)

The a processes sleep for three seconds, wake up and call b:x(), and then print the result.

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

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

loop(Tag) ->
    sleep(),
    Val = b:x(),
    io:format("Vsn1 (~p) b:x() = ~p~n",[Tag, Val]),
    loop(Tag).

sleep() ->
    receive
       after 15000 -> true
    end.
```

Example: Start up scenario (3)

Start up scenario: Compile b and a and start a couple of a processes:

```
2> c(b).
{ok,b}
3> c(a).
{ok,a}
4> a:start(one).
<0.53.0>
Vsn1 (one) b:x() = 1
5> a:start(two).
<0.55.0>
Vsn1 (one) b:x() = 1
Vsn1 (two) b:x() = 1
Vsn1 (one) b:x() = 1
Vsn1 (two) b:x() = 1
Vsn1 (one) b:x() = 1
Vsn1 (two) b:x() = 1
```

Example: Change and recompile b (4)

Scenario: Go into the editor and change b to the following:

```
-module(b).
-export([x/0]).
x() -> 2.
```

We recompile b in the shell. This is what happens:

```
6> c(b).
{ok,b}
Vsn1 (two) b:x() = 2
Vsn1 (one) b:x() = 2
Vsn1 (two) b:x() = 2
...
```

The two original versions of a are still running, but they call the new version of b. When we call b:x() from within the module a, we really call the latest version of b.

Example: Change and recompile a (5)

Scenario: Go into the editor and change the version a from

```
io:format("Vsn1 (~p) b:x() = ~p~n",[Tag, Val]),
```

into

```
io:format("Vsn2 (~p) b:x() = ~p~n",[Tag, Val]),
```

Now we compile and start a new a and the result is:

Example: ... and the result is (6)

```
7> c(a).
{ok,a}
Vsn1 (one) b:x() = 2
Vsn1 (two) b:x() = 2
8> a:start(three).
<0.67.0>
Vsn1 (one) b:x() = 2
Vsn1 (two) b:x() = 2
Vsn2 (three) b:x() = 2
Vsn1 (one) b:x() = 2
Vsn1 (two) b:x() = 2
Vsn2 (three) b:x() = 2
...
```

When we start the new version of a, we see that new version running. The existing processes running the first version of a are still running that old version.

Example: Change and recompile b again (7)

Scenario: Go into the editor and change b to the following:

```
-module(b).
-export([x/0]).
x() -> 3.
```

We recompile b in the shell. This is what happens:

```
9> c(b).
{ok,b}
Vsn1 (one) b:x() = 3
Vsn1 (two) b:x() = 3
Vsn2 (three) b:x() = 3
....
```

Both the old and new versions of a call the latest version of b.

Example: Change and recompile a Vsn3 (7)

Scenario: Recompile a. into a new `Vsn3` and start a new a.

```
10> c(a).
{ok,a}
11> a:start(four).
<0.79.0>
Vsn2 (three) b:x() = 3
Vsn3 (four) b:x() = 3
Vsn2 (three) b:x() = 3
Vsn3 (four) b:x() = 3
...
```

The output contains the last two versions of a (versions 2 and 3); the process running version 1 of a's code has died.

Erlang can have two versions of a module running at any one time, the current version and an old version.

That's all!