Erlang and Concurrency

Erlang is a concurrent functional programming language. It was initially designed at the Ericsson Computer Science Laboratory by Joe Armstrong and his team and was used for various system implementations inside Ericsson. It was released as open-source Erlang in 1998 under EPL (derivative of MPL). Here we are going to present at first an overview of Erlang and its basic features, show the basic programming expressions and then talk about Erlang and concurrency. Finally we are going to elaborate on how Erlang performs on multicore systems.

General Characteristics

To understand the features of Erlang as a new programming language, we should consider what kind of problems its designers wanted to address. Erlang was designed for implementing telecom switching systems. These systems are large and complex, with huge amount of code which is in general difficult to maintain. They are soft-real time systems where performance and timed response is important. They need to handle a large number of parallel operations that may be distributed over a network of nodes. Finally they need to be reliable and perform in a predicted "logical" way at all times even in the presence of errors (either hardware or software). We can see that these requirements are not restricted only to those systems but can be seen in many other software systems.

Erlang addresses these requirements by doing work that is commonly done by the operating system. It provides a lightweight mechanism to create and destroy processes that can communicate with each other through the exchange of simple messages. Each process has its own environment and it is isolated from all other processes - it can be viewed as an isolated virtual machine that evaluates its own erlang code. The only way to communicate with a process is through sending a message to it. That's why each one has a unique name - identifier that other processes can use to send their messages. These processes can run on a single node or on multiple since there is no share state making the distribution of the system easier. The message passing is asynchronous to maintain the isolation of the processes. A message can be delivered or not and the only way to know is to ask the recipient if it got the message.
The isolation of processes is the mechanism that favors the creation of a reliable fault tolerant system given that a software error that is present in a particular process will not affect the rest of the parallel processes that are running at the same time. In addition Erlang provides clever mechanisms to catch and handle these errors even when happening remotely. This means that there are mechanism for a process to monitor another process and in the case of an error/failure, the monitoring process gets notified with the error and can handle it appropriate.

Another feature of erlang is the fact that it is platform independent. Erlang code is compiled into intermediate code by the erlang compiler (called BEAM) which then is executed by the erlang runtime environment. There is also the HiPE compiler (High Performance Erlang) implemented and maintained by the Upsala University that compiles Erlang into native code if higher performance is required. Finally Erlang supports on-line code updates meaning that the system's code can be modified without stopping and restarting the system, minimizing its down time.

**Erlang Basics**

Erlang is a single - assignment, dynamically typed functional language. Single assignment means that a variable gets bound to a particular value and can not be reassigned to different value. This means that erlang code has no side effect only eager evaluation of erlang expressions. This is a very important point both for more clear programming and easier debugging of programs since you can easily find where the wrong assignment in an error happened and for concurrency implementation since there is no shared memory. (This is partial true since there are some data structures that can be modified and shared by multiple erlang processes which are in memory or on disk hash table-like data structures called ETS and DETS and those should be used with caution by the programmer.)

Erlang data types:
- Integers (of arbitrary size): 1234568789, 16#AB10F
- Floating point numbers: 12,456789345
- Atoms (like enum types with global scope of other languages - distinguished values) : red,white,'my atom', '2'
- Tuples (group a fixed number of values in a single entity): {name,bob}, {age,26}, {person, {{name,bob}, {age,26}}} (a tuple that contains an atom and another tuple.
- Lists (group variable number of values in a single entity): [1,2,3,4,5] list of five integers, [H|T] H is a variable representing the first list element,T represents a list of the remaining elements, "Hello" represents [72,101,108,108,111] strings are just list of the ascii representation of the character in the string.
• Funs (functions): fun(X) -> X*2 end
• Pids (unique process identifiers)
• Refs (guaranteed unique identifiers)

Erlang basic expressions:
• Pattern matching: (LHS) = (RHS), the RHS is evaluated and matched against LHS, any unbounded variables are bound in this process (unification).

  \[ \{\text{Name, Age}\}_{\text{Rest}} = \{\{\text{bob,25}\},\{\text{mary,30}\},\{\text{alice,18}\}\} \] . This pattern matching succeeds and the following bindings are happening Name=bob, Age=25, Rest = \[\{\{\text{mary,30}\},\{\text{alice,18}\}\}\] 
  \{_\_\_\_Height\} = \{15,25,30\} . Use of the anonymous variable for values that are not important.

• Functions: Head1 -> Body1; Head2 -> Body2;....Headn -> Bodyn, Head is the name of the function (which is practically an atom) followed by a set of arguments (patterns) and followed optionally by a guard (see below). Body is a comma separated list of expressions. Named functions are defined inside modules and are called combining the module name with the function name. Functions can be used as arguments to other functions or returned by other functions.

  \%
  define module geometry\%
  module(geometry).
  \%
  export area function outside of this module\%
  export([area/1])

  \%
  define area function\%
  area({rectangle,Width,Height}) -> Width*Height;
  area({circle,Radius}) -> Radius*Radius.

  \%
  can be called as: \%
  geometry:area(circle,4). geometry:area(rectangle,4,5).

  \%
  sequential map function - example of function as argument\%
  map(_, []) ->[];
  map(F, [H,T]) -> [F(H),map(F,T)].

• Guard: when P , where P is a predicate that can evaluate to true or false, they are used at the head of a function or at a pattern matching evaluation.

max(X,Y) when X>Y -> Y;
max(Y) -> Y.
List comprehensions: \([X \mid \text{Qualifier1, Qualifier2, \ldots}]\), where \(X\) is an expression and \(\text{Qualifier}\) is either a generator or a filter. Generator is of the form \(\text{Pattern} \leftarrow \text{ListExpr}\), and filter is predicate that evaluates to true/false.

\([2 \times X \mid X \leftarrow L]\) %doubles each element in the list \(L\%

%return a list of all permutations of the elements of a list%

\[
\text{perm}([]) \rightarrow [[]];
\text{perm}(L) \rightarrow [H|T \mid H \leftarrow L, T \leftarrow \text{perms}(L-\{H\})]
\]

**Concurrency Oriented Programming**

The creator of Erlang, Joe Armstrong, introduces the term Concurrency Oriented Programming (COP) to describe a programming paradigm where you model the problem in question as a set of parallel operations and the term Concurrent Oriented Programming Languages (COPL) to describe the languages that provide concurrency as part of their implementation. (The COP paradigm is based on the Communicating Sequential Processes which influenced also the occam programming language) He argues that people are used to concurrency since our world is concurrent and in general we, as human beings, are performing many parallel tasks at the same time and not sequential tasks, so programming in a parallel way should come natural in a language that provides the right mechanisms, like Erlang. What we have to do is identify the parallel tasks (in a right granularity for our problem), identify the tasks that require communication between them (message channels) and write the different messages that are required to be exchange between these tasks.

**Concurrent Programming in Erlang**

In order to control a set of parallel activities Erlang has primitives for multiprocessing: spawn starts a parallel computation (called a process); send sends a message to a process; and receive receives a message from a process.

\(\text{spawn} / 3\) starts execution of a parallel process and returns an identifier which may be used to send messages to and receive messages from the process. Pids are used for all forms of communication with a process.

\[
\text{Pid} = \text{spawn} (\text{Module}, \text{FunctionName}, \text{ArgumentList})
\]

The call to \(\text{spawn} / 3\) returns immediately when the new process has been created and does not wait for the given function to evaluate.
Inter-process Communication

In Erlang the only form of communication between processes is by message passing. A message is sent to another process by the primitive `!' (send):

\[ \text{Pid ! Message} \]

Pid is the identifier of the process to which Message is sent. A message can be any valid Erlang term. send is a primitive which evaluates its arguments. Its return value is the message sent.

Sending a message is an asynchronous operation so the send call will not wait for the message either to arrive at the destination or to be received. Even if the process to which the message is being sent has already terminated the system will not notify the sender. This is in keeping with the asynchronous nature of message passing - the application must itself implement all forms of checking. Messages are always delivered to the recipient, and always delivered in the same order they were sent.

The primitive receive is used to receive messages. It has the following syntax:

\[ \begin{array}{l}
\text{receive} \\
\quad \text{Message1 [when Guard1]} \rightarrow \\
\quad \quad \text{Actions1} ; \\
\quad \text{Message2 [when Guard2]} \rightarrow \\
\quad \quad \text{Actions2} ; \\
\quad \ldots \\
\end{array} \]

end

Each process has a mailbox and all messages which are sent to the process are stored in the mailbox in the same order as they arrive. In the above, Message1 and Message2 are patterns which are matched against messages that are in the process's mailbox. When a matching message is found and any corresponding guard succeeds the message is selected, removed from the mailbox and then the corresponding ActionsN are evaluated. receive returns the value of the last expression evaluated in the actions. As in other forms of pattern matching, any unbound variables in the message pattern become bound. Any messages which are in the mailbox and are not selected by receive will remain in the mailbox in the same order as they were stored and will be matched against in the next receive. The process evaluating receive will be suspended until a message is matched.

Erlang has a selective receive mechanism, thus no message arriving
unexpectedly at a process can block other messages to that process. However, as any messages not matched by receive are left in the mailbox, it is the programmer's responsibility to make sure that the system does not fill up with such messages.

**Receiving messages from a specific process**

We often want to receive messages from a specific process. To do this the sender must explicitly include its own process identifier in the message:

```
Pid ! {self(),abc}
```

which sends a message that explicitly contains the sender’s process identifier. The built-in function self() returns the identifier of the calling process. This could be received by:

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

If Pid is bound to the sender's process identifier then evaluating receive as above would receive messages only from this process.

**Counter Example**

The following program is a module which creates processes containing counters which can be incremented.

```
-module(counter).
-export([start/0,loop/1,increment/1,value/1,stop/1]).

%% First the interface functions.
start() ->
  spawn(counter, loop, [0]).
increment(Counter) ->
  Counter ! increment.
value(Counter) ->
  Counter ! {self(),value},
  receive
    {Counter,Value} ->
      Value
  end.
```

stop(Counter) ->
    Counter ! stop.

%% The counter loop.
loop(Val) ->
    receive
        increment ->
            loop(Val + 1);
        {From, value} ->
            From ! {self(), Val},
            loop(Val);
        stop -> % No recursive call here
            true;
        Other -> % All other messages
            loop(Val)
    end.

The counter process uses the selective receive mechanism to process the incoming requests. It also presents a solution to the problem of handling unknown messages. The last clause in the receive has the unbound variable Other as its message pattern; this will match any message which is not matched by the other clauses. Here we ignore the message and continue by waiting for the next message. This is the standard technique for dealing with unknown messages: receive them to get them out of the mailbox.

When we access the value of a counter, we must send our Pid as part of the message to enable the counter process to send back a reply. This reply also contains the identifier of the sending process, in this case the counter, to enable the receiving process specifically to wait for the message containing the reply. It is unsafe just to wait for a message containing an unknown value, in this case a number, as any other message which happens to be sent to the process will be matched. Messages sent between processes, therefore, usually contain some way of identifying them, either by their contents, as in the request messages to the counter process, or by including some `unique' and easily recognisable identifier, as in the reply to the value request.

**Registered Processes**

In order to send a message to a process, one needs to know its identifier (Pid). In some cases this is neither practical nor desirable: for example, in a
large system there may be many global servers, or a process may wish to hide its identity for security reasons. To allow a process to send a message to another process without knowing its identity we provide a way to register processes, i.e. to give them names. The name of a registered process must be an atom.

**Basic primitives**

Four built-in functions are provided for manipulating the names of registered processes:

- **register(Name, Pid)**  
  Associates the atom Name with the process Pid.

- **unregister(Name)**  
  Removes the association between the atom Name and a process.

- **whereis(Name)**  
  Returns the process identifier associated with the registered name Name. If no processes have been associated with this name, it returns the atom undefined.

- **registered()**  
  Returns a list of all the currently registered names.

The message sending primitive `!' also allows the name of a registered process as a destination. For example

```
number_analyser ! {self(), {analyse,[1,2,3,4]}}
```

means send the message `{Pid,{analyse,[1,2,3,4]}}` to the process registered as number_analyser. Pid is the processes identifier of the process evaluating send.

**Client-Server Model**

A major use of registered processes is to support programming of the client-server model. In this model there is a server, which manages some resource, and a number of clients which send requests to the server to access the resource, as illustrated in Figure 5.4. Three basic components are necessary to implement this model - a server, a protocol and an access library. We illustrate the basic principles by some examples.

The following example is a server which could be used in a telephone
exchange to analyse telephone numbers dialled by users of the exchange. start() creates a number analyser server process by calling spawn and then registers the server process as number_analyser. The server process then loops in the function server and waits for service requests. If an \{add_number,Seq,Dest\} request is received the new number sequence is added to the lookup table along with the destination to return if this sequence is analysed. This is done by the function insert. The requesting process is sent the message ack. If the request \{analyse,Seq\} is received then number analysis is performed on the sequence Seq by calling lookup. A message containing the result of the analysis is sent to the requesting process. We do not give the definitions of the functions insert and lookup as they are not important to this discussion.

The request message sent to the server by the client contains the Pid of the client. This makes it possible to send a reply to the client. The reply message sent back to the client also contains a `sender', the registered name of the server, allowing the client process to receive the reply message selectively. This is safer than just waiting for the first message to arrive - the client process may already have some messages in the mailbox or another process may have sent it a message before the server replies.

We have now written the server and defined the protocol. We have decided to implement a synchronous protocol here, in which there will always be a reply to each request made to the server. In the reply from the server we give the `sender' as number_analyser, the registered name of the server, not wishing to disclose the Pid of the server. We now define interface functions to access the server in a standard manner. The functions add_number and analyse implement the client's side of the protocol described above. They both use the local function request to send the request and receive the reply.

```
-module(number_analyser).
-export([start/0,server/1]).
-export([add_number/2,analyse/1]).

start() ->
    register(number_analyser, spawn(number_analyser, server, [nil])).

%% The interface functions.
add_number(Seq, Dest) ->
    request({add_number,Seq,Dest}).
analyse(Seq) ->
    request({analyse,Seq}).
request(Req) ->
    number_analyser ! {self(), Req},
```

receive
  {number_analyser,Reply} ->
    Reply
  end.

%%% The server.
server(AnalTable) ->
  receive
    {From, {analyse,Seq}} ->
      Result = lookup(Seq, AnalTable),
      From ! {number_analyser, Result},
      server(AnalTable);
    {From, {add_number, Seq, Dest}} ->
      From ! {number_analyser, ack},
      server(insert(Seq, Dest, AnalTable))
  end.

Creation of processes and message passing benchmark:
We tried to measure how fast processes are created and how a large number of exchanges of a message is handled in Erlang. We did that by implementing a basic algorithm for the ring test: Creating N nodes in a ring and forwarding m times a message through each node, exchanging in the end N*M messages. This is not a very parallel program or algorithm since at any given time only one process is doing some work (forwarding the message) but it does shows that a truly large number of messages is handled efficiently by Erlang. Following are the results we got for various values of N and M and the erlang code of our implementation.

Measurements:
100 nodes 100 iterations, 10000 messages: System time 19 ms, CPU time 20 ms
100 nodes 1000 iterations, 100000 messages: System time 118 ms, CPU time 110 ms
1000 nodes 1000 iterations, 1000000 messages: System time 1095 ms, CPU time 1660 ms
1000 nodes 5000 iterations, 5000000 messages: System time 5265 ms, CPU time 5110 ms
5000 nodes 5000 iterations, 25000000 messages: System time 26521 ms, CPU time 25610 ms

-module(ring2).
-export([[timeTest/3,start/3]]).
%measure the elapsed system and cpu time%
timeTest(N,M,Message) ->
    statistics(runtime),
    statistics(wall_clock),
    start(N,M,Message),
    {_,TCPU} = statistics(runtime),
    {_,Time} = statistics(wall_clock),
    io:format("Time: ~p ms ~n, CPUTime: ~p ms ~n", [Time,TCPU])
.

%start ring test of N processes and M cycles of passing the Message%
start(N,1,Message) -> firstNode(N,1,Message);
start(N,M,Message) ->
    Next = firstNode(N,M,Message),
    sendLoop(M-1,Next).

%FirstNode handling which creates the first forwarding node
and sends the first message.%
firstNode(N,M,Message) ->
    First = self(),
    Next = spawn(fun() -> launchNode(N-1,M,First) end),
    Next ! Message,
    Next.

%creation of forwarding nodes.%
launchNode(1,M,StartNode) -> sendLoop(M,StartNode);
launchNode(N,M,StartNode) ->
    Next = spawn(fun() -> launchNode(N-1,M,StartNode) end),
    sendLoop(M,Next).

%the loop performed by each node%
sendLoop(1,Pid) -> send(Pid);
sendLoop(M,Pid) ->
    send(Pid),sendLoop(M-1,Pid).

%send the msg that is received to the process Pid.%
send(Pid) ->
    receive Message ->
        Pid ! Message
    end.

**Multicore Programming and Erlang.**

The current implementations of the Erlang runtime environment have support for executing on SMP architectures. This means that the ERE can schedule parallel processes on different CPU's taking advantage of multi-core systems. This support has been added since May 2006 and Ericsson released a Telephone Gateway Controller, which used the SMP Erlang almost a year
afterwards. The gains were considered satisfiable even though the transition and the benchmark testing was done in a dual-core AMD64 machine from a single core Power-PC.

Because of the way that concurrency is an integral part of the language, erlang programs are supposed to have significant performance gains when running on a multi-core system, possibly n times faster on an n-core processor. However this is not always true. Benchmarking [1] has shown that performance gains are limited by how parallel the program is and by the current ERE implementation even on adequately parallel programs. In the "Bing Bang" test where 1000 processes are spawned and exchange messages with each other, benchmarking has shown that impressive speedups can be achieved up to 32 cores and more cores than that cause a slow down to the program. This can be explained by the fact that ERE implementation does use threads and synchronization primitives to handle the parallel Erlang processes so contention is does occur when dealing with so many threads. More specific, there is a contention of schedulers trying to access the same queue for fetching erlang processes. What is more, there are programs that have showed very bad results when running on multi-core environments. These are programs that have at some point less running parallel processes than the cores that the SMP/Erlang is scheduled to run on, then the ERE schedulers are idle and put to sleep by the operating system and waked up again. This process and the system calls to the OS extremely slow down the program and produce very bad results.

In general there should not be any difference for the programmer between programming in a single core environment and in a multi core environment. But considering the above points, there are some subtle differences and the programmer should take these points into consideration when addressing a particular problem. In general, to gain in performance we must have "enough" parallel processes doing some work for the solution of the problem. These processes ideally have no side-effects and must reduces as much as possible the sequential access to resources i.e. IO or if possible parallelize the access to that resources.

So let's see as an example the map function of a list. The sequential design of this function is the following:

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

A possible parallel approach to the same problem will be to create N parallel processes where N is the length of the list, each of them executing F on an element of the list. We can see the code below:

```erlang
pmap1(F, L) -> S = self(),
Ref = erlang:make_ref(),
foreach(fun(I) -> spawn(fun() -> do_f1(S, Ref, F, I) end) end, L),
```

```erlang
do_f1(S, Ref, F, I) -> erlang:recv(Ref),
F(I),
```

```erlang
foreach(fun() -> do_f1(S, Ref, F, I) end) end, L),
```

%%% gather the results
gather1(length(L), Ref, []).

do_f1(Parent, Ref, F, I) -> Parent ! {Ref, (catch F(I))}.

gather1(0, _, L) -> L;
gather1(N, Ref, L) -> receive
    {Ref, Ret} -> gather1(N-1, Ref, [Ret|L])
end.

We can see here that the semantics of the parallel map are not exactly the same with the ones of the sequential map. If an exception occurs the error value of the exception will be added to the return list but the sequential program would have terminated in this case. We can write more code to provide the same semantics but more complexity will be added. Also we assume here that the map function does not cause any side effects so we assume that we have the same results as sequential mapping. Ideally for our example, the map function should be a heavy computation function and the size of the list should be adjusted to the number of cores we're running on in order to have the best results.

So programming in a parallel way is again more complex and should be judged by what we want to perform and if we will have gains from parallelizing our program. In either case, parallelizing sequential operations isn’t as straightforward as it may seem, and Erlang doesn’t always yield good result. The idea behind SMP Erlang is to speed up programs that have naturally concurrent patterns.

References

Making reliable distributed systems in the presence of software errors, Joe Armstrong, November 2003.


HiPE on AMD-64, Daniel Luna, Mikaerl Pettersson, Konstantinos Sagonas.

Erlang Programming for Multi-core presentation by Ulf Wiger, Ericsson AB.[1]