Parallel and Distributed Programs
Lecture F12
24 February, 2014

Joe Armstrong

February 23, 2014

Contents

1 Introduction

In this lecture we'll look at some ways of writing *potentially parallel* programs.

In Erlang we write concurrent programs. The algorithms in the lecture will run sequentially if run on a single core machine, but will run in parallel if run on a multi-core computer. That's why they are called "potentially parallel."

To make a parallel program either the programmer must say what is to be done in parallel, or the system must discover this automatically.

Automatic parallelisation of programs has been studied for many years. It is very good for certain classes of problem and very bad for other classes of problem.

1.1 GPU Parallism

GPU = Graphics Programming Unit. All the computations are the same.

For example: To compute the temperature distribution over a conducting surface we divide the surface into a grid and repeatedly average the temperatures at the four corners of the squares surrounding each of the points on the grid.

```
             |
             X (i,j+1)
             |
             |
----X--------X-------X----
  (i-1,j)  (i,j)   (i+1,j)
             |
             |
             X (i,j-1)
             |
             |


  T[k+1] = (T[k][i,j+1] + T[k][i-1,j] +
            T[k][i+1,j] + T[k][i,j-1])/4
```

Erlang is not suited for this.

1.2 Embarrassing Parallelism

The nature of the problem is naturally parallel and there are a lot of things to be done. There is little or no communication between the tasks. The parallelism is obvious and inherent in the problem.

Example: A web server with millions of simultaneous connected users. Erlang is very good at this.

1.3 Declarative Parallelism

Sometimes called "functional parallelism". In a pure functional programming language `f(g(...), h(...))` `g(...)` and `h(...)` can be evaluated in any order and in parallel.

Unfortunately, while this is true, it is still a research problem how to map the computation onto the hardware we have *today*, and *hardware changes in time* so the solution we find today will not work tomorrow.

There have been a large number of attempts to build parallel hardware to do this. None have provided lasting solutions.

2 Expressing Parallel Behaviour

Different languages have syntactic constructs for expressing parallel behavior, or can do this algorithmically. This section reviews some commonly used methods for expressing parallel behavior.

2.1 ParBegin .. ParEnd

```
ParBegin
   x;
   y;
   z;
ParEnd
```

The above expression means evaluate `x`, `y` and `z` in parallel.

`Fork` ... `Join` is also used.

If shared memory is involved, simultaneous access to the shared memory must be avoided. If no shared memory is involved the computations can proceed in parallel.

Both `ParBegin` .. `ParEnd` and `Fork` ... `Join` can be implemented using the lower level primitives `promise` and `yield`.

2.2 Promises

A "*Promise*" is a commitment to deliver a value in the future.

```
    P
    |
    |  promise(F)
T1  +------>-------+
    |              |
    |              |
    |              |
    +------<-------+
    |
    |
T2  +
    |
```

At some time `T1` a process `P` wants to compute `F()` so it calls `promise(F)` which returns a tag `Tag`. `P` continues, and `F()` is computed in parallel. At some time later (`T2`) when `P` wants the return value it calls `yield(Tag)`. `yield` blocks until the result of the computation is available.

Promises can be implmeneted in Erlang as follows:

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

test1() -> fib(40).

test2() -> promise(fun() -> fib(40) end).

fib(1) -> 1;
fib(2) -> 1;
fib(N) -> fib(N-1) + fib(N-2).

promise(F) ->
    Tag = make_ref(),
    S = self(),
    spawn(fun() -> S ! {Tag, F()} end),
    Tag.

query(Tag) ->
    receive
        {Tag, Reply} ->
            {finished, Reply}
    after 0 ->
            not_finished
    end.

yield(Tag) ->
    receive
        {Tag, Reply} ->
            Reply
    end.
```

`promises:test1()` blocks until the result is available:

```
> promises:test1().
.... yawn ..
...  shell is blocked ...
... I can't do anything ...
165580141
```

Using a promise we can do *useful things* ® while the promised computation is being performed:

```
2> Tag = promises:test2().
Ref<0.0.0.104>
3> promises:query(Tag).
not_finished
4> 213813813861736*128763816381638163.
27531482667950334890792863030968
5> promises:query(Tag).
not_finished
6> 19379127391273971239719379173979372193137*18763861238183638.
363627257286948472579176034410575587459129627874409292406
7> promises:query(Tag).
{finished,165580141}
```

Note: If a command fails in the shell, the promise will not be honoured, the results will be send back to the old shell which has died!

2.3 Broken Promises

What happens if `P` dies *Before* the worker has computed its response? Should the worker be killed?

What happens if the worker dies while computing F. Who should be told? Who should be killed?

Distributed Computing is the art of repairing broken promises.

2.4 Maps

Here is a simple implement of `smap` (Sequential map) and `pmap` (Parallel map).

```
-module(maps).
-export([pmap/2, smap/2]).

smap(F, L) -> [F(I) || I <- L].

pmap(F, L) ->
    S = self(),
    Pids = [spawn(fun() ->
                           S ! {self(),catch F(I)}
                   end) || I <- L],
    gather_replies(Pids).

gather_replies([Pid|T]) ->
    receive
        {Pid, Val} -> [Val|gather_replies(T)]
    end;
gather_replies([]) ->
    [].
```

In the non-failure case smap and pmap1 behave identically.

```
1> Double = fun(X) -> 2*X end.
#Fun<erl_eval.6.71889879>
2> maps:smap(Double,[1,2,3,4]).
[2,4,6,8]
3> maps:pmap(Double,[1,2,3,4]).
[2,4,6,8]
```

Now let's try and double the list [1,2,3,a,3,4,b]. In the failure case smap will fail on the first error (ie when computing 2*a) but pmap will fail for every bad element in the list.

```
4> maps:smap(Double,[1,2,a,3,4,b]).
** exception error: an error occurred when evaluating an
   arithmetic expression
     in operator  */2
        called as 2 * a
     in call from maps:'-smap/2-lc$^0/1-0-'/2 (maps.erl, line 5)
     in call from maps:'-smap/2-lc$^0/1-0-'/2 (maps.erl, line 5)
5> maps:pmap(Double,[1,2,a,3,4,b]).
[2,4,
 {'EXIT',{badarith,[{erlang,'*',[2,a],[]},
                    {maps,'-pmap/2-fun-0-',3,
                          [{file,"maps.erl"},{line,10}]}]}},
 6,8,
 {'EXIT',{badarith,[{erlang,'*',[2,b],[]},
                    {maps,'-pmap/2-fun-0-',3,
                          [{file,"maps.erl"},{line,10}]}]}}]

```

2.5 Broken Maps

The idea of pmap seems simple *but it is not*. Here are some of the problems that might occur.

A computation in the map fails
: Assume one of the computations of F(I) fails. We could kill all the other computations. But what happens if we don't really care if the computation failed? Many of the earlier computations might have succeeded so why drop the results? The computations might have taken a long time, it would be a shame to loose the results.

We have too much concurrency

`pmap` will be *too parallel* if we have a list of 1M processes and the function is *Double* then pmapping the list would be wrong. We'd create too many processes.

The overheads are too great

It might turn out that creating a parallel computation costs more than just doing the sequential computation.

Discussion: `map` and `pmap` have identical behavior when there are no errors. In `pmap` we fail immediately on the first error. In `pmap` we cannot easily fail on the first error. (We could link all the parallel processes together, and let them all die if any one dies)

2.6 Map-Reduce

Map-reduce is the name given to a method of parallelising computations. Note: The word "map" does not mean the same as the word map used in functional programming.

A full map-reduce framework provides the following:

- A way of starting a number of parallel computations.
- A number of parallel proceses which perform computations. The map processes performs some computation and sends a stream of `{Key,Value}` results to a reduce process.
- A reduce process which merges values with the same Key.

For example: Calculating the frequency of words in a large set of documents. Assume the number of documents is `K`.

Here's how we might do this in Erlang:

- We spawn a reducer process, Call this process `Reducer`.
- We spawn `K` parallel processes (workers) , and give each process the name of a file to work on, and the process identifier `Reducer`.
- Each worker analyses the data in the file it was given and sends a sequence of `{Word,Count}` pairs to the reducer process. For example, if the word "tuple" occurred ten times in a document it would send a `{<<"tuple">>,10}` tuple to the reduce process.
- The reducer process merges the word count streams. So for example if it has received the tuples `{<<"tuple">>,10}` and `{<<"tuple">>,15}` it will know that the number of times the word has been seen is 25.

Several practical issues have to be dealt with when implementing map-reduce:

- Termination. How do we know when the map phase has terminated. Should the reducer wait for all map processes to terminate before it terminates.
- Bottlenecks. The reduce process could become a bottleneck. All the results eventually go to a single process. We might want to avoid this.

In practice we would probably limit the time for a map-reduce operation to a fixed time, and build a tree of reducers. In the word-count example we could build two reducers (one for words beginning `a..m` and `n..z` or 26 reducer processes for letters starting `a`, `b` and so on.

3 Components

Parallel systems are often build from compents. Here are afew examples:

3.1 Server

A server has state. When it receives requests from a client, it performs a computation which might update the state and then sends a message back to the client.

(This is the server in the so call Client-Server Model)

3.2 Publisher

A publisher has a list of channels. `[{Channel, [Pid1, Pid2, ...]}]` The publisher accepts four input messages.

`{subscribe,Channel,Pid}`
: Adds `Pid` to the list of pids associated with `ChannelName`. Creates a new channel if a channel does not exist.

`{unsubscript_all,Pid}`
: removes all subscriptions that Pid has.

`{unsubscript_all,Channel, Pid}`
: Remove the `Pid` to `Channel`.

`{publish,Channel,Msg}`
: Sends Msg to all the subscribers associated with `Channel`. It is not an error to send a message to a channel with zero subscribers.

Publisher has no errors.

A publisher is the PUB in the so called PUB-SUB model.

Note: PUB-SUB is an enormously important model: Twitter, Facebook, Mailing Lists, IRC Newsletters are all variants on PUB-SUB

Actually PUB-SUB is a confusing term. It could mean the names of two different processes. A Publisher Process that talks to a Subscriber Process. Or, it could mean the name of two different message sent to a publisher process.

3.3 Dealer

A dealer has `N` output ports. Messages sent to the input ports are round-robbined to the output ports.

`{add_output,Pid}`
: add Pid to the list of output ports.

`{send,Message}`
: Send `Message` to one of the output ports.

  errors: There are no output ports

3.4 Router

A router has an internal routing table which is a list of pairs of the form. `[{Tag,OutPort}]`. Message sent to the input port must have a tag, these messages are sent to the appropriate output port.

`{add_route,Tag,Pid}`
: Adds a routing table entry `{Tag, Pid}` to the routing table.

  errors: Tag already used.

`{unroute,Tag}`
: Removes a routing table entry with `Tag` from the routing table.

`{send,Tag,Msg}`
: Sends `Msg` to the process assocated with `Tag`.

  errors: No process associated with Tag

3.5 Middle Man

A middle man translates between two different messaging formats. It has two ports, call these `a` and `b`. Messages to port `a` are in format `one` and messages to port `b` are in format `two`.

Messages arriving at port `a` should be converted from format `one` to format `two` and sent to port `b` and *vice versa.*

Here's a middle man that converts bwteen Metric and US meaasurements:

```
               +-----+
               |     |
    {km,3}   A |     |B  {miles,1.864}
-------->------+     +------->---------
               |     |
               +-----+
```

And in the opposite direction:

```
                 +-----+
                 |     |
    {km,3.218} A |     |B  {miles,2}
  -------<-------+     +-------<---------
                 |     |
                 +-----+
```

Examples: Convert JSON messages to XML messages, or Erlang terms. Convert SAX events to XML terms.

4 Router Code

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

start(Name) ->
    register(Name, spawn(fun() ->
                                 put(name, Name),
                                 router()
                         end)).

router() ->
    receive
        {do,{add_route,Tag,Pid},Epid} ->
            case get(Tag) of
                undefined -> put(Tag, Pid);
                _         -> Epid ! {eBadTag, get(name), Tag}
            end;
        {do, {route, Tag, Msg}, Epid} ->
            case get(Tag) of
                undefined -> Epid ! {ebadRoute, get(name), Tag};
                Pid       -> Pid ! Msg
            end;
        {do, {remove_route, Tag}} ->
            erase(Tag)
    end,
    router().

```

The router assumes all messages are are of the form: `{do,Cmd,Epid}` If the command works it does what it is supposed to do. If it fails it sends a message to `Epid`

5 Publisher Code

Here's a simple PUB-SUB publisher process. There is no error handling.

```
-module(pubsub0).
-compile(export_all).

start() ->
    stop(),
    register(pubsub0, spawn(pubsub0, init, [])).

stop() ->
    (catch exit(whereis(pubsub0), kill)),
    ok.

publish(G, M) -> pubsub0 ! {to_group,G,{publish, M}}.

subscribe(G) -> pubsub0 ! {to_group,G,{subscribe, self()}}.

init() ->
    pubsub_loop([]).

pubsub_loop(L) ->
    receive
        {to_group, Group, Msg} ->
            case find_group(Group, L) of
                {yes, Pid} ->
                    Pid ! Msg,
                    pubsub_loop(L);
                no ->
                    Pid = spawn(pubsub, agent_loop, [Group,[]]),
                    Pid ! Msg,
                    L1 = [{Group,Pid}|L],
                    pubsub_loop(L1)
            end
    end.

agent_loop(Group, Pids) ->
    receive
        {subscribe, My_pid} ->
            agent_loop(Group, [My_pid|Pids]);
        {publish, Msg} ->
            [Pid ! Msg || Pid <- Pids],
            agent_loop(Group, Pids)
    end.

```

```
find_group(Group, L) ->
    case lists:keysearch(Group, 1, L) of
        {value,{_,Val}} -> {yes, Val};
        false           -> no
    end.
```