Beautiful Concurrency with Erlang

Kevin Scaldeferri

OSCON

23 July 2008

6 years at Yahoo, building large high-concurrency distributed systems

Not an expert, don’t use it professionally

Dabbled, liked it, want to share what I think is cool

What is Erlang?

- Strict pure functional language
- Strong dynamic typing
  - weak structural user-defined types
- Interpreted
- Syntax similar to Prolog & ML
- Concurrency primitives
- Created at Ericsson for telecom applications in 1987

Not going to talk about syntax, basic language features, etc

Go to Francesco Cesarini's talk yesterday.

Erlang Concurrency Primitives

- spawn - create a process
- ! - send a message to a process
- receive - listen for a message

Parallelizing Algorithms

- Quicksort
- Shamelessly stolen from http://21ccw.blogspot.com/2008/05/parallel-quicksort-in-erlang-part-ii.html

```
qsort([]) -> [];
qsort([Pivot|Rest]) ->
  qsort([ X || X <- Rest, X < Pivot]) ++ [Pivot] ++ qsort([ Y || Y <- Rest, Y >= Pivot]).
```

Erlang: one of those quicksort in 3 lines languages
but... to small to read

```
qsort([]) -> [];
qsort([Pivot|Rest]) ->
      qsort([ X || X <- Rest,
                   X < Pivot])
  ++ [Pivot]
  ++ qsort([ Y || Y <- Rest,
                   Y >= Pivot]).
```

```
qsort([]) -> [];
qsort([Pivot|Rest]) ->
  Left = [ X || X <- Rest,
                X < Pivot],
  Right = [ Y || Y <- Rest,
                 Y >= Pivot],
  qsort(Left) ++ [Pivot]
    ++ qsort(Right).
```

Extract temp variables

```
qsort([]) -> [];
qsort([Pivot|Rest]) ->
  Left = [ X || X <- Rest,
                X < Pivot],
  Right = [ Y || Y <- Rest,
                 Y >= Pivot],
  [SortedLeft, SortedRight] =
    map(fun qsort/1, [Left, Right]),
  SortedLeft ++ [Pivot]
    ++ SortedRight.
```

Add a map(), which looks odd but now we're ready to do some magic

```
qsort([]) -> [];
qsort([Pivot|Rest]) ->
  Left = [ X || X <- Rest,
                X < Pivot],
  Right = [ Y || Y <- Rest,
                 Y >= Pivot],
  [SortedLeft, SortedRight] =
    pmap(fun qsort/1, [Left, Right]),
  SortedLeft ++ [Pivot]
    ++ SortedRight.
```

Now we're running on as many cores as you've got

Who thinks this is a good idea?

Don’t try this at home

actually 10x slower on my machine
spawning a process is fast, but still much slower than a comparison / list cons
a better example - web spidering

```
spider(URIs) ->
    ...
    Links = pmap(fun get_links/1,
                 URIs),
    ...
```

web spider needs to fetch content, parse XML/HTML, extract links
Significant speedup here, both from parallelizing network requests and CPU

```
pmap(F, L) ->
  S = self(),
  Pids = map(fun(I) -> spawn(fun() -> pmap_f(S, F, I)
end) end, L),
  pmap_gather(Pids).

pmap_f(Parent, F, I) ->
  Parent ! {self(), (catch F(I))}.

pmap_gather([H|T]) ->
  receive
    {H, Ret} -> [Ret|pmap_gather(T)]
  end;
pmap_gather([]) -> [].
```

```
pmap(F, L) ->
  S = self(),
  Pids = map(fun(I) -> spawn(fun() -> pmap_f(S, F, I)
end) end, L),
  pmap_gather(Pids).

pmap_f(Parent, F, I) ->
  Parent ! {self(), (catch F(I))}.

pmap_gather([H|T]) ->
  receive
    {H, Ret} -> [Ret|pmap_gather(T)]
  end;
pmap_gather([]) -> [].
```

pmap uses map

```
pmap(F, L) ->
  S = self(),
  Pids = map(fun(I) -> spawn(fun() -> pmap_f(S, F, I)
end) end, L),
  pmap_gather(Pids).

pmap_f(Parent, F, I) ->
  Parent ! {self(), (catch F(I))}.

pmap_gather([H|T]) ->
  receive
    {H, Ret} -> [Ret|pmap_gather(T)]
  end;
pmap_gather([]) -> [].
```

but instead of running the function directly, spawns a new process to run it

```
pmap(F, L) ->
  S = self(),
  Pids = map(fun(I) -> spawn(fun() -> pmap_f(S, F, I)
end) end, L),
  pmap_gather(Pids).

pmap_f(Parent, F, I) ->
  Parent ! {self(), (catch F(I))}.

pmap_gather([H|T]) ->
  receive
    {H, Ret} -> [Ret|pmap_gather(T)]
  end;
pmap_gather([]) -> [].
```

```
pmap(F, L) ->
  S = self(),
  Pids = map(fun(I) -> spawn(fun() -> pmap_f(S, F, I)
end) end, L),
  pmap_gather(Pids).

pmap_f(Parent, F, I) ->
  Parent ! {self(), (catch F(I))}.

pmap_gather([H|T]) ->
  receive
    {H, Ret} -> [Ret|pmap_gather(T)]
  end;
pmap_gather([]) -> [].
```

apply the function to the list item in the child process

```
pmap(F, L) ->
  S = self(),
  Pids = map(fun(I) -> spawn(fun() -> pmap_f(S, F, I)
end) end, L),
  pmap_gather(Pids).

pmap_f(Parent, F, I) ->
  Parent ! {self(), (catch F(I))}.

pmap_gather([H|T]) ->
  receive
    {H, Ret} -> [Ret|pmap_gather(T)]
  end;
pmap_gather([]) -> [].
```

then send it back to the parent

```
pmap(F, L) ->
  S = self(),
  Pids = map(fun(I) -> spawn(fun() -> pmap_f(S, F, I)
end) end, L),
  pmap_gather(Pids).

pmap_f(Parent, F, I) ->
  Parent ! {self(), (catch F(I))}.

pmap_gather([H|T]) ->
  receive
    {H, Ret} -> [Ret|pmap_gather(T)]
  end;
pmap_gather([]) -> [].
```

parent gathers results

```
pmap(F, L) ->
  S = self(),
  Pids = map(fun(I) -> spawn(fun() -> pmap_f(S, F, I)
end) end, L),
  pmap_gather(Pids).

pmap_f(Parent, F, I) ->
  Parent ! {self(), (catch F(I))}.

pmap_gather([H|T]) ->
  receive
    {H, Ret} -> [Ret|pmap_gather(T)]
  end;
pmap_gather([]) -> [].
```

receive a message from each Pid we spawned

```
pmap(F, L) ->
  S = self(),
  Pids = map(fun(I) -> spawn(fun() -> pmap_f(S, F, I)
end) end, L),
  pmap_gather(Pids).

pmap_f(Parent, F, I) ->
  Parent ! {self(), (catch F(I))}.

pmap_gather([H|T]) ->
  receive
    {H, Ret} -> [Ret|pmap_gather(T)]
  end;
pmap_gather([]) -> [].
```

cons up the return values

```
pmap(F, L) ->
  S = self(),
  Pids = map(fun(I) -> spawn(fun() -> pmap_f(S, F, I)
end) end, L),
  pmap_gather(Pids).

pmap_f(Parent, F, I) ->
  Parent ! {self(), (catch F(I))}.

pmap_gather([H|T]) ->
  receive
    {H, Ret} -> [Ret|pmap_gather(T)]
  end;
pmap_gather([]) -> [].
```

```
pmap(F, L) ->
  S = self(),
  Pids = map(fun(I) -> spawn(fun() -> pmap_f(S, F, I)
end) end, L),
  pmap_gather(Pids).

pmap_f(Parent, F, I) ->
  Parent ! {self(), (catch F(I))}.

pmap_gather([H|T]) ->
  receive
    {H, Ret} -> [Ret|pmap_gather(T)]
  end;
pmap_gather([]) -> [].
```

Distributed Systems

Who uses Twitter? Who's frustrated by twitter?
Who's written their own twitter clone?

Twitter

“Twitter is, fundamentally, a messaging system. Twitter was not architected as a messaging system, however. For expediency's sake, Twitter was built with technologies and practices that are more appropriate to a content management system.” -Alex Payne

Erlang approach: treat it as a messaging application. Model users by processes sending messages to each other.

```
create_user(Name) ->
    User = #user{name=Name},
    Pid = spawn(fun() -> loop(User) end),
    try register(Name, Pid) of
        true -> {ok, Pid}
    catch
        error:badarg ->
            exit(Pid, in_use),
            {error, in_use}
    end.
```

```
create_user(Name) ->
    User = #user{name=Name},
    Pid = spawn(fun() -> loop(User) end),
    try register(Name, Pid) of
        true -> {ok, Pid}
    catch
        error:badarg ->
            exit(Pid, in_use),
            {error, in_use}
    end.
```

create a user record

```
create_user(Name) ->
    User = #user{name=Name},
    Pid = spawn(fun() -> loop(User) end),
    try register(Name, Pid) of
        true -> {ok, Pid}
    catch
        error:badarg ->
            exit(Pid, in_use),
            {error, in_use}
    end.
```

spawn a new process to manage the user

```
create_user(Name) ->
    User = #user{name=Name},
    Pid = spawn(fun() -> loop(User) end),
    try register(Name, Pid) of
        true -> {ok, Pid}
    catch
        error:badarg ->
            exit(Pid, in_use),
            {error, in_use}
    end.
```

register a name for the process, so we can send using the username rather than pid

```
follow(UserPid, OtherName) ->
    send(UserPid, {follow, OtherName}).
...

send(Name, Msg) ->
    try Name ! Msg
    catch
        error:badarg ->
            {error, no_such_user}
    end.
```

```
follow(UserPid, OtherName) ->
    send(UserPid, {follow, OtherName}).
...

send(Name, Msg) ->
    try Name ! Msg
    catch
        error:badarg ->
            {error, no_such_user}
    end.
```

to add a follower

```
follow(UserPid, OtherName) ->
    send(UserPid, {follow, OtherName}).
...

send(Name, Msg) ->
    try Name ! Msg
    catch
        error:badarg ->
            {error, no_such_user}
    end.
```

send a message to the user

```
follow(UserPid, OtherName) ->
    send(UserPid, {follow, OtherName}).
...

send(Name, Msg) ->
    try Name ! Msg
    catch
        error:badarg ->
            {error, no_such_user}
    end.
```

saying “follow that guy”

```
follow(UserPid, OtherName) ->
    send(UserPid, {follow, OtherName}).
...

send(Name, Msg) ->
    try Name ! Msg
    catch
        error:badarg ->
            {error, no_such_user}
    end.
```

```
follow(UserPid, OtherName) ->
    send(UserPid, {follow, OtherName}).
...

send(Name, Msg) ->
    try Name ! Msg
    catch
        error:badarg ->
            {error, no_such_user}
    end.
```

send is just a thin wrapper around ! with error handling

Going Global

- Distribute across multiple machines?
- Just use global names

so far, just running on one machine (can handle tens of thousands, maybe hundreds, of users)
eventually need to grow past that to multiple machines.
Fortunately this is easy

```
create_user(Name) ->
  User = #user{name=Name},
  Pid = spawn(fun() -> loop(User) end),
  try register(Name, Pid) of
    true -> {ok, Pid}
  catch
    error:badarg ->
      exit(Pid, in_use),
      {error, in_use}
  end.
```

just change register

```
create_user(Name) ->
  User = #user{name=Name},
  Pid = spawn(fun() -> loop(User) end),
  try global:register_name(Name, Pid) of
    true -> {ok, Pid}
  catch
    error:badarg ->
      exit(Pid, in_use),
      {error, in_use}
  end.
```

to global register

```
create_user(Name) ->
  User = #user{name=Name},
  Pid = spawn(fun() -> loop(User) end),
  try global:register_name(Name, Pid) of
    true -> {ok, Pid}
  catch
    error:badarg ->
      exit(Pid, in_use),
      {error, in_use}
  end.
```

```
send(Name, Msg) ->
  try Name ! Msg
  catch
    error:badarg ->
      {error, no_such_user}
  end.
```

similarly, change !

```
send(Name, Msg) ->
  try global:send(Name, Msg)
  catch
    error:badarg ->
      {error, no_such_user}
  end.
```

to global:send

```
send(Name, Msg) ->
  try global:send(Name, Msg)
  catch
    error:badarg ->
      {error, no_such_user}
  end.
```

Reliable Distributed Systems

What if a process crashes?

OTP

Open Telecom Platform

OTP provides frameworks for common application patterns, and handles reliability by watching and restarting processes

```
-module(twitterl).

-behaviour(gen_server).
```

We'll use the gen_server behaviour (similar to a Java interface)

```
create_user(Name) ->
  gen_server:start_link(
    {global, Name},
    ?MODULE,
    [#user{name=Name}],
    []
  ).
```

start_link handles registering names, spawning the process and running the main loop

```
create_user(Name) ->
  gen_server:start_link(
    {global, Name},
    ?MODULE,
    [#user{name=Name}],
    []
  ).
```

```
create_user(Name) ->
  gen_server:start_link(
    {global, Name},
    ?MODULE,
    [#user{name=Name}],
    []
  ).
```

using global names again

```
create_user(Name) ->
  gen_server:start_link(
    {global, Name},
    ?MODULE,
    [#user{name=Name}],
    []
  ).
```

required callbacks provided by the current module

```
create_user(Name) ->
  gen_server:start_link(
    {global, Name},
    ?MODULE,
    [#user{name=Name}],
    []
  ).
```

initial state

```
follow(UserName, OtherName) ->
  gen_server:call(
    {global, UserName},
    {follow, OtherName}
  ).
post(UserName, Msg) ->
  gen_server:call(
    {global, UserName},
    {post, Msg}
  ).
```

other API function follow a common pattern

```
follow(UserName, OtherName) ->
  gen_server:call(
    {global, UserName},
    {follow, OtherName}
  ).
post(UserName, Msg) ->
  gen_server:call(
    {global, UserName},
    {post, Msg}
  ).
```

```
follow(UserName, OtherName) ->
  gen_server:call(
    {global, UserName},
    {follow, OtherName}
  ).
post(UserName, Msg) ->
  gen_server:call(
    {global, UserName},
    {post, Msg}
  ).
```

make a call to the server

```
follow(UserName, OtherName) ->
  gen_server:call(
    {global, UserName},
    {follow, OtherName}
  ).
post(UserName, Msg) ->
  gen_server:call(
    {global, UserName},
    {post, Msg}
  ).
```

using the global name

```
follow(UserName, OtherName) ->
  gen_server:call(
    {global, UserName},
    {follow, OtherName}
  ).
post(UserName, Msg) ->
  gen_server:call(
    {global, UserName},
    {post, Msg}
  ).
```

“follow” message

```
handle_call({follow, Other},
            _From, State) ->
  NewF = [Other|State#user.following],
  gen_server:call(
    {global, Other},
    {add_follower, State#user.name}),
  {reply, ok,
    State#user{following=NewF}};
```

set up callbacks for expected messages

```
handle_call({follow, Other},
            _From, State) ->
  NewF = [Other|State#user.following],
  gen_server:call(
    {global, Other},
    {add_follower, State#user.name}),
  {reply, ok,
    State#user{following=NewF}};
```

to follow another user

```
handle_call({follow, Other},
            _From, State) ->
  NewF = [Other|State#user.following],
  gen_server:call(
    {global, Other},
    {add_follower, State#user.name}),
  {reply, ok,
    State#user{following=NewF}};
```

add them to the list of people we're following

```
handle_call({follow, Other},
            _From, State) ->
  NewF = [Other|State#user.following],
  gen_server:call(
    {global, Other},
    {add_follower, State#user.name}),
  {reply, ok,
    State#user{following=NewF}};
```

call the other process

```
handle_call({follow, Other},
            _From, State) ->
  NewF = [Other|State#user.following],
  gen_server:call(
    {global, Other},
    {add_follower, State#user.name}),
  {reply, ok,
    State#user{following=NewF}};
```

```
handle_call({follow, Other},
            _From, State) ->
  NewF = [Other|State#user.following],
  gen_server:call(
    {global, Other},
    {add_follower, State#user.name}),
  {reply, ok,
    State#user{following=NewF}};
```

and tell them to add you as a follower

```
handle_call({follow, Other},
            _From, State) ->
  NewF = [Other|State#user.following],
  gen_server:call(
    {global, Other},
    {add_follower, State#user.name}),
  {reply, ok,
    State#user{following=NewF}};
```

tell gen_server all is good, and the new state

```
handle_call({add_follower, F},
            _From, State) ->
  NewF = [F | State#user.followers],
  {reply, ok,
    State#user{followers=NewF}};
```

the other process adds you to their follower list

```
handle_call({post, Msg},
            _From, State) ->
  map(fun(Name) ->
        gen_server:cast(Name,
{posted, State#user.name, Msg})
      end,
      State#user.followers),
  {reply, ok, State};
```

to post a message

```
handle_call({post, Msg},
            _From, State) ->
  map(fun(Name) ->
        gen_server:cast(Name,
{posted, State#user.name, Msg})
      end,
      State#user.followers),
  {reply, ok, State};
```

for each follower

```
handle_call({post, Msg},
            _From, State) ->
  map(fun(Name) ->
        gen_server:cast(Name,
{posted, State#user.name, Msg})
      end,
      State#user.followers),
  {reply, ok, State};
```

send the message we posted

```
handle_call({post, Msg},
            _From, State) ->
  map(fun(Name) ->
        gen_server:cast(Name,
{posted, State#user.name, Msg})
      end,
      State#user.followers),
  {reply, ok, State};
```

use cast() because we don’t care about any reply

```
handle_cast({posted, Other, Msg},
            State) ->
  %% really store in DB, SMS, etc
  io:fwrite("~s ~s~n",
            [Other, Msg]),
  {noreply, State};
```

just for illustrative purposes, print messages we receive.
Really stick it in DB, send to SMS, etc

In Summary

Erlang makes adding concurrency to your programs nearly trivial

Thank You

Resources

- http://erlang.org/
- http://trapexit.org/
- #erlang on freenode
- Erlang-questions@erlang.org
- CEAN