

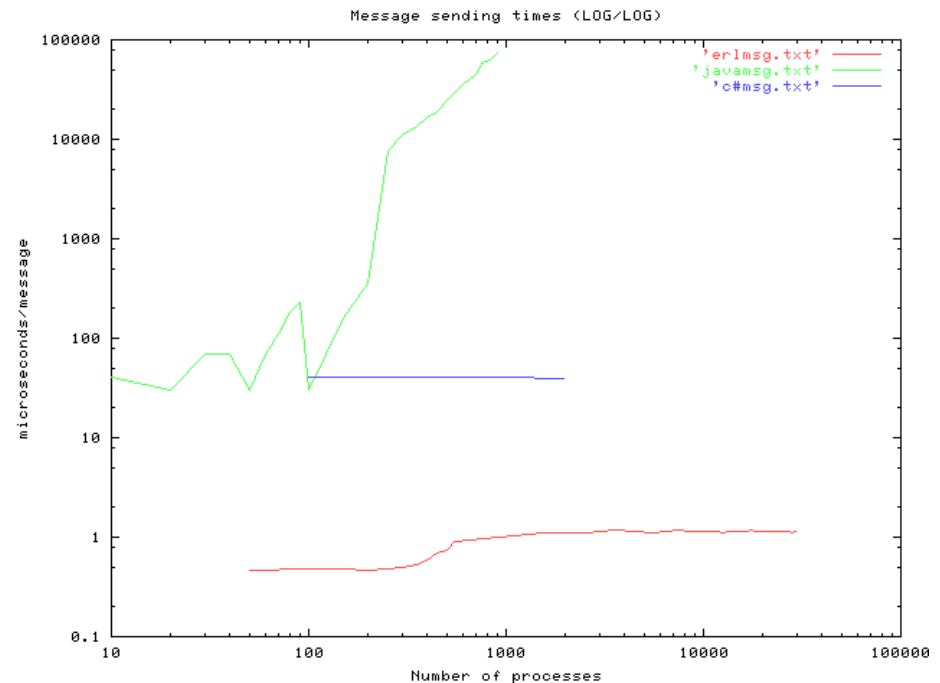
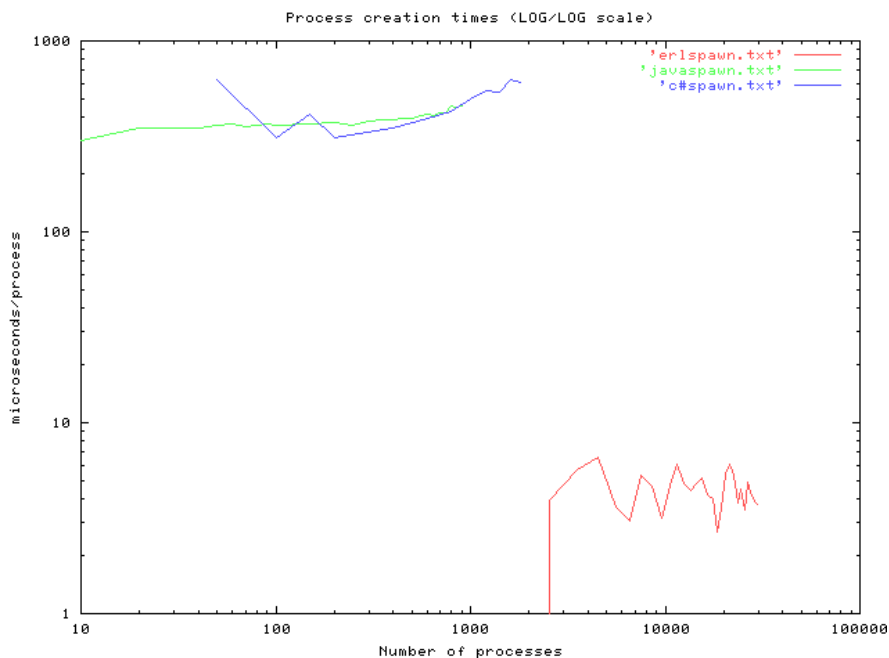
Principles of Concurrency and Parallelism

Lecture 5: Erlang

1/31/12

Example

- Suppose we have N threads (or processes, tasks) that form a ring
- Each thread communicates with its neighbor forwarding a message
- How big can we make the ring?
- How long does it take to send a message?



CS390C: Principles of Concurrency and Parallelism

Philosophy

- Independent processes
 - suitable for executing on distributed machines
- No sharing
 - (Deep) copy data sent on messages
 - no cross-machine pointers
 - no locks, data races, synchronization issues, ...
- All processes have a unique name
- Asynchronous sends, synchronous receives
 - Eventual delivery
 - But if A sends messages m1 and m2 to B, m2 will never arrive before m1

Key features

- Functional
 - single assignment (every variable assigned to at most once)
- Lightweight first-class processes
- Pattern-matching
- Small collection of datatypes
 - lists, tuples, pairs
- Dynamic typing
- Realtime concurrent garbage collection

Examples

```
-module(math).  
-export([fac/1]).
```

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

```
> math:fac(25).  
15511210043330985984000000
```

```
lookup(Key, {Key, Val, _, _}) ->  
    {ok, Val};  
lookup(Key, {Key1, Val, S, B}) when Key < Key1 ->  
    lookup(Key, S);  
lookup(Key, {Key1, Val, S, B}) ->  
    lookup(Key, B);  
lookup(Key, nil) ->  
    not_found.
```

Examples

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

```
sort([Pivot|T]) ->  
  sort([X||X <- T, X < Pivot]) ++  
  [Pivot] ++  
  sort([X||X <- T, X >= Pivot]);  
sort([]) -> [].
```

```
> Adder = fun(N) -> fun(X) -> X + N end end.  
#Fun  
> G = Adder(10).  
#Fun  
> G(5).  
15
```

Concurrency

```
-module(m).  
-export([loop/0]).  
loop() ->  
    receive  
        who_are_you ->  
            io:format("I am ~p~n", [self()]),  
            loop()  
    end.  
1> P = spawn(m, loop, []).  
<0.58.0>  
2> P ! who_are_you.  
I am <0.58.0>  
who_are_you
```

Concurrency

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

```
start() ->  
    spawn(counter, loop, [0]).
```

```
loop(Val) ->  
    receive  
        increment -> loop(Val + 1)  
end.
```

Issues:

- Cannot directly access counter value.
- Messaging protocol is explicit (via message increment)

Refinement

```
-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.  
  
loop(Val) ->  
    receive  
        increment -> loop(Val + 1);  
        {From,value} -> From ! {self(),Val}, loop(Val);  
        stop -> true;  
        Other -> loop(Val)  
    end.
```

Concurrency

```
-module(M).  
-export([start/0, ping/1, pong/0]).  
  
ping(0) ->  
    pong ! finished,  
    io:format("ping finished~n", []);  
ping(N) ->  
    pong ! {ping, self()},  
    receive pong ->  
        io:format("Ping received pong~n", [])  
    end,  
    ping(N - 1).  
  
pong() ->  
    receive  
        finished -> io:format("Pong finished~n", []);  
        {ping, Ping_PID} ->  
            io:format("Pong received ping~n", []),  
            Ping_PID ! pong,  
            pong()  
    end.  
  
start() -> register(pong, spawn(M, pong, [])),  
           spawn(M, ping, [3]).
```

Distributed Programming

- Can generalize previous example to a distributed environment

```
-module(M).  
-export([start/0, ping/1, pong/0]).
```

```
ping(0,Pong_node) ->  
    {pong, Pong_Node} ! finished,  
    io:format("ping finished~n", []);  
ping(N) ->  
    {pong, Pong_Node} ! {ping, self()},  
    receive pong ->  
        io:format("Ping received pong~n", [])  
    end,  
    ping(N - 1, Pong_Node).
```

```
pong() ->  
    receive  
        finished -> io:format("Pong finished~n", []);  
        {ping, Ping_PID} ->  
            io:format("Pong received ping~n", []),  
            Ping_PID ! pong,  
            pong()  
    end.
```

```
start_pong() -> register(pong, spawn(M, pong, [])),  
start_ping(Pong_Node) -> spawn(M, ping, [3, Pong_Node]).
```

On one host: `erl -sname ping`

On another: `erl -sname pong`

On one node:

`M:start_pong()`.

On another:

`M:start_ping(pong@<host>)`.

Monitoring

```
...
process_flag(trap_exit, true),
Pid = spawn_link(fun() -> ... end),
receive
    {'EXIT', Pid, Why} ->
        ...
end
```

Client/Server

```
server(Fun, Data) ->
  receive
    {new_fun, Fun1} ->
      server(Fun1, Data);
    {rpc, From, ReplyAs, Q} ->
      {Reply, Data1} = Fun(Q, Data),
      From ! {ReplyAs, Reply},
      server(Fun, Data1)
  end.
```

```
rpc(A, B) ->
  Tag = new_ref(),
  A ! {rpc, self(), Tag, B},
  receive
    {Tag, Val} -> Val
  end
```

Concurrency Patterns

Unicast

A ! B

RPC call

```
Call (RPC)
  A ! {self(), B},
  receive
    {A, Reply} ->
      Reply
  end
```

Event Handling

```
receive A -> A end
```

Callback

```
receive
  {From, A} ->
    From ! F(A)
end
```

Concurrency Patterns

Callback within RPC

```
A ! {Tag, X},  g(A, Tag).
```

```
g(A, Tag) ->  
  receive  
    {Tag, Val} -> Val;  
    {A, X} ->  
      A ! F(X),  
      go(A, Tag)  
  end.
```

Timeouts

```
receive
  Message1 [when
    Actions1 ;
  Message2 [when
    Actions2 ;
  ...
after
  TimeoutExpr ->
    ActionsT
end
```

```
get_event() ->
  receive
    {mouse, click} ->
      receive
        {mouse, click} ->
          double_click
        after double_click_interval()
          -> single_click
      end ...
  end.
```

Readings

Concurrent Programming in Erlang (Part I)

<http://www.erlang.org/download/erlang-book-part1.pdf>

See also: www.erlang.org