Scenes From the Language Struggle in Toronto, Ontario

Mike Nolta

Ch-Ch-Changes

- OpenMP & MPI are fine for now, but we need language support for parallelism.
- Corbato's Law: "The number of lines of code a programmer can write in a fixed amount of time is the same independent of the language used."

Two roads diverged

- C & Fortran were designed decades ago.
  - They're going to have to adapt, or be replaced by new languages.
- Example of adaptation: C-blocks
- Example new language: Erlang

C-blocks

What are blocks?

- Developed by Apple as part of Snow Leopard.
- Called “closures” in most other languages.
- Can think of them as improved function pointers.

Similar to function ptr...

```
// function pointer
int (*fptr)( int i );

int j = fptr(3);

// block
int (^blck)( int i );

int j = blck(3);
```

...but defined inline

```
// function pointer
int (*fptr)( int i );
fptr = &function_name;
int j = fptr(3);

// block
int (^blck)( int i );
blck = ^(int i) { return i+2; };
int j = blck(3);
```

Blocks are functions w/ bound variables

```
blck = ^(int i) { return i+2; };
int j = blck(3); // 5
```

```
int n = 30;
blck = ^(int i) { return i+n; };
int j = blck(3); // 33
```

Bound variables are constant

```
int n = 30;
blck = ^(int i) { return i+n; };
```

```
int j = blck(3); // 33
n = -789;
int k = blck(3); // 33
```

Bound variables are constant (2)

```
int n = 30;
blck = ^(int i) {
     n = 4; // illegal
     return i+n;
};
```

What does this have to do w/ parallelism?

```
for ( i = 0; i < N; i++ ) {
    result[i] = do_work(data, i);
}
```

Grand Central

```
dispatch_apply( N, queue,
      ^(size_t i) {
         result[i] = do_work(data, i);
      }
   );
```

Erlang

What is Erlang?

- Erlang is a "new" computer language created by Ericsson ~20 yrs ago.
  - It's new in the sense that it hasn't received a lot of attention until recently.
- Originally designed for telephony hardware.
- Open sourced in 1998.

Who uses it?

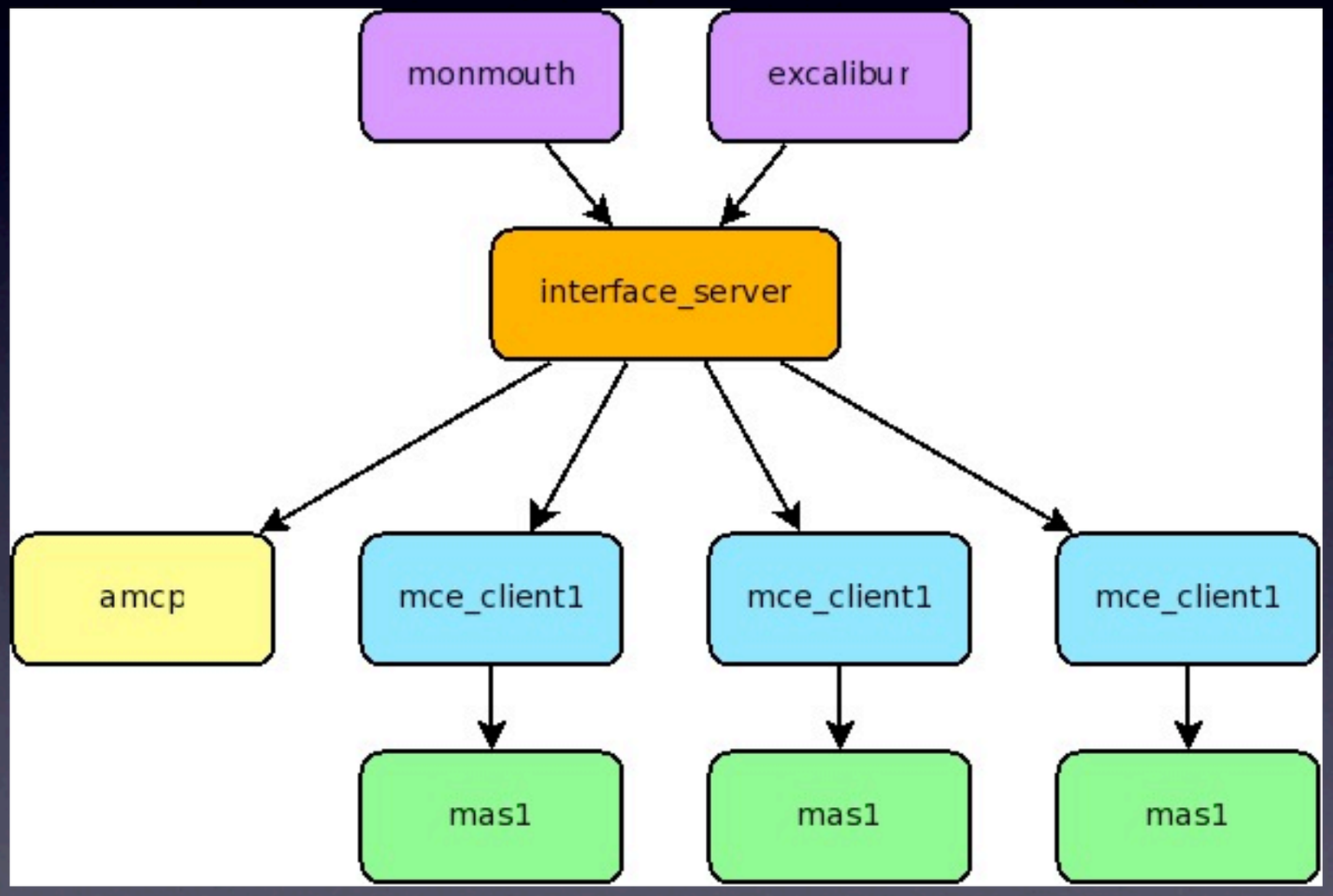
- Ericsson, of course, in their telephony equipment.
  - Systems have achieved 99.9999999% uptime (0.03 sec downtime per year).
- Facebook's chat servers are partially written in Erlang; handles 70 million users.
- Amazon's "SimpleDB" service; rumored that IMdb is going to use Erlang.

Why use it?

- Designed to build massively concurrent, distributed, fault-tolerant systems.

Why I'm interested

ACT control software



Basic syntax

Types

- Atoms: purely symbolic, have no value
  - examples: `true`, `ok`, `not_ready`
- Integers, doubles
  - integers are unbounded
- Binaries

Collections

- Tuples
  - e.g.: {error, badarg}
- Linked lists
  - e.g.: [1,2,3,4]
- Strings (e.g., "hello") are really just lists of integers

Naming conventions

- Variables have to start with an uppercase letter.
- If it starts with a lowercase letter it's an atom.
- Example: true is an atom, True is a variable.

Erlang is Functional

“Functional”

- A “functional” language treats computation as the evaluation of functions, and doesn’t have mutable state.
- Other functional languages: Lisp, ML, Mathematica.
- The are no loops in Erlang, just recursive functions.

Simple functions

```
function_name( arg1, arg2, … ) ->
    statement1,
    statement2,
    …
    statementN.
```

Example

```
area_of_rect( Width, Height ) ->
    Width*Height.
```

Single assignment

```
[nolta@richelieu pca]$ erl
1> X = 3.
3
2> X = 4.
** exception error: no match of
right hand side value 4
```

Erlang is Declarative

Pattern-matching

- "=" in Erlang is not an assignment operator, but a *pattern matching* operator.

```
1> {X,Y} = {ok,56}.
{ok,56}
2> X.
ok
3> Y.
56
```

Pattern-matching

- "=" in Erlang is not an assignment operator, but a *pattern matching* operator.

```
1> [Head|Tail] = [1,2,3,4].
[1,2,3,4]
2> Head.
1
3> Tail.
[2,3,4]
```

Functions, too

```
fibonacci( 0 ) ->
    0;
fibonacci( 1 ) ->
    1;
fibonacci( N ) ->
    fibonacci(N-1)*fibonacci(N-2).
```

Functions, too

```
fibonacci( 0 ) ->
    0;
fibonacci( 1 ) ->
    1;
fibonacci( N ) when is_integer(N) and N>1 ->
    fibonacci(N-1)*fibonacci(N-2).
```

Example: summation

```
sum( List ) ->
    sum( List, 0 ).

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

Example: building a list

```
range( N ) ->
    range( N, [] ).

range( 0, R ) ->
    R;
range( I, R ) ->
    range( I-1, [I|R] ).
```

```
1> range(5).
[1,2,3,4,5]
```

Erlang is Concurrent

Actor model

- Actors can:
  - create new actors,
  - send messages to other actors,
  - receive messages.
- All communication is asynchronous.

Creating processes

```
Pid = spawn( function ).
Pid = spawn( node, function ).
```

- *function* runs concurrently in its own process.
- Shares no state with parent (or any other) process.
- Pid is unique process ID.

Sending messages

```
Pid ! {list_sum,[1,4,5]}.
```

- Sends the message "`{list_sum, [1,4,5]}`" asynchronously to the process Pid.
- Messages can be anything.
- **Doesn't matter if Pid is local or remote process.**

Receiving messages

```
receive
    {list_sum,List} ->
        X = sum(List);
    {list_mult,List} ->
        X = mult(List)
end.
```

- Every process has its own mailbox queue of messages.
- receive blocks until a message arrives matching one of its pattern.

Example: RPC

```
Pid ! {rpc,method,Args},
receive
    {reply,Pid,Reply} ->
        io:format( “~p~n”, [Reply] )
end.
```

Best of both worlds

- Erlang processes are similar to MPI processes: no shared state, pass messages back and forth.
- But Erlang processes are lightweight like OpenMP threads; only costs ~300 bytes to create one.

1000's of processes

- Random blog quote: "The best I got on my MacBook Pro after numerous runs was 0.301 seconds with 2400 processes."

“Process oriented”

- Languages like C++/Python are *object-oriented*: state is encapsulated in classes.
- Erlang is *process-oriented*: state is encapsulated in processes.

Full example

Wikipedia MPI “hello world”

```
/*
  "Hello World" Type MPI Test Program
 */
 #include <mpi.h>
 #include <stdio.h>
 #include <string.h>

 #define BUFSIZE 128
 #define TAG 0

 int main(int argc, char *argv[])
 {
   char idstr[32];
   char buff[BUFSIZE];
   int numprocs;
   int myid;
   int i;
   MPI_Status stat;

   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
   MPI_Comm_rank(MPI_COMM_WORLD,&myid);

 if(myid == 0)
    {
      printf("%d: We have %d processors\n", myid, numprocs);
      for(i=1;i<numprocs;i++)

      {
        sprintf(buff, "Hello %d! ", i);
        MPI_Send(buff, BUFSIZE, MPI_CHAR, i, TAG, MPI_COMM_WORLD);
      }
      for(i=1;i<numprocs;i++)
      {
        MPI_Recv(buff, BUFSIZE, MPI_CHAR, i, TAG, MPI_COMM_WORLD, &stat);
        printf("%d: %s\n", myid, buff);
      }
    }
    else
    {
      /* receive from rank 0: */
      MPI_Recv(buff, BUFSIZE, MPI_CHAR, 0, TAG, MPI_COMM_WORLD, &stat);
      sprintf(idstr, "Process %d ", myid);
      strcat(buff, idstr);
      strcat(buff, "reporting for duty");
      /* send to rank 0: */
      MPI_Send(buff, BUFSIZE, MPI_CHAR, 0, TAG, MPI_COMM_WORLD);
    }

    MPI_Finalize();
    return 0;
 }
```

Output

```
[nolta@tpb5 pca]$ mpirun C ./a.out
0: We have 8 processors
0: Hello 1! Process 1 reporting for duty
0: Hello 2! Process 2 reporting for duty
0: Hello 3! Process 3 reporting for duty
0: Hello 4! Process 4 reporting for duty
0: Hello 5! Process 5 reporting for duty
0: Hello 6! Process 6 reporting for duty
0: Hello 7! Process 7 reporting for duty
[nolta@tpb5 pca]$
```

Master process

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

main(N) ->
    Slaves = make_slaves(N),
    collect_replies(Slaves).
```

Making the slaves

```
make_slave( N )->
    Slave = spawn( fun() -> slave(N) end ),
    Slave ! { hi, self(),
        io_lib:format("Hello ~w!",[N]) },
    {Slave,N}.

make_slaves( N ) ->
    lists:map( fun(I) -> make_slave(I) end, range(N) ).
```

Slave process

```
slave(I) ->
    receive
        {hi, From, Message} ->
            Response = io_lib:format(
                "~s Process ~w reporting for duty",
                [Message,I] ),
            From ! { howdy, self(), Response }
end.
```

Collecting the replies

```
collect_replies( [{Pid,N}|T] ) ->
    receive
        { howdy, Pid, Message } ->
            io:format( "0: ~s~n", [Message] ),
            collect_replies(T);
    end;
collect_replies( [] ) ->
    done.
```

```
Eshell V5.6.2  (abort with ^G)
1> hello_world:main(7).
0: Hello 1! Process 1 reporting for duty
0: Hello 2! Process 2 reporting for duty
0: Hello 3! Process 3 reporting for duty
0: Hello 4! Process 4 reporting for duty
0: Hello 5! Process 5 reporting for duty
0: Hello 6! Process 6 reporting for duty
0: Hello 7! Process 7 reporting for duty
done
```

What happens if a slave dies?

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

```
main(N) ->
    Slaves = make_slaves(N),
    {ThirdPid,_} = lists:nth(3,Slaves),
    exit( ThirdPid, bye_bye ),
    collect_replies(Slaves).
```

```
Eshell V5.6.2  (abort with ^G)
1> hello_world2:main(7).
0: Hello 1! Process 1 reporting for duty
0: Hello 2! Process 2 reporting for duty

BREAK: (a)bort (c)ontinue (p)roc info (i)nfo (l)oaded
       (v)ersion (k)ill (D)b-tables (d)istribution
a
```

Restarting the slave

```
collect_replies( [{Pid,N}|T] ) ->
    receive
        { howdy, Pid, Message } ->
            io:format( "0: ~s~n", [Message] ),
            collect_replies(T);
        after 5000 ->
            io:format( "restarting slave~w~n", [N] ),
            exit( Pid, too_late ),
            NewSlave = make_slave(N),
            collect_replies( [NewSlave|T] )
    end;
collect_replies( [] ) ->
    done.
```

```
Eshell V5.6.2  (abort with ^G)
1> hello_world3:main(7).
0: Hello 1! Process 1 reporting for duty
0: Hello 2! Process 2 reporting for duty
restarting slave3
0: Hello 3! Process 3 reporting for duty
0: Hello 4! Process 4 reporting for duty
0: Hello 5! Process 5 reporting for duty
0: Hello 6! Process 6 reporting for duty
0: Hello 7! Process 7 reporting for duty
done
```

Restarting the slave

```
collect_replies( [{Pid,N}|T] ) ->
    receive
        { howdy, Pid, Message } ->
            io:format( "0: ~s~n", [Message] ),
            collect_replies(T);
        {'DOWN',_,process,Pid,Reason} ->
            io:format( "restarting slave~w because ~w~n",
                [N,Reason] ),
            NewSlave = make_slave(N),
            collect_replies( [NewSlave|T] )
    end;
collect_replies( [] ) ->
    done.
```

```
Eshell V5.6.2  (abort with ^G)
1> hello_world4:main(7).
0: Hello 1! Process 1 reporting for duty
0: Hello 2! Process 2 reporting for duty
restarting slave3 because bye_bye
0: Hello 3! Process 3 reporting for duty
0: Hello 4! Process 4 reporting for duty
0: Hello 5! Process 5 reporting for duty
0: Hello 6! Process 6 reporting for duty
0: Hello 7! Process 7 reporting for duty
done
```

Only scratched the surface

Hot code swapping

```
server( Module ) ->
    receive
        { apply, From, Args } ->
            From ! Module:handle( Args ),
            server( Module );
        { hotswap, NewModule } ->
            server( NewModule )
end.
```

“Batteries included”

- Comes with a bunch of useful stuff:
  - debuggers, profilers, tracers, coverage, process monitors, …
  - Mnesia: a distributed parallel database
  - OTP: standard library for building fault-tolerant applications

Numerical Work

Erlang is not ready for numerical work

- Numerical libraries are essentially non-existent.
- But this will change in the next few years.
- It will be a lot easier to add BLAS/LAPACK/FFT/etc to Erlang than to make other languages concurrent.

Summary

- Erlang: functional, declarative, concurrent, fault-tolerant.
- Worth keeping an eye on.

More information

- http://erlang.org/ for the source code.
- “Progamming Erlang” book by Joe Armstrong (one of the original creators).
- Google “erlang movie” -- hilarious.