# Processes and Messages

Mark Greenstreet

CpSc 418 – Jan. 9, 2017

Outline:

## Objectives

- Introduce Erlang's features for concurrency and parallelism
  - Spawning processes.
  - Sending and receiving messages.
- Describe timing measurements for these operations and the implications for writing efficient parallel programs.
  - **Communication often dominates the runtime of parallel programs.**
- The source code for the examples in this lecture is available here: procs.erl.

## Processes – Overview

- The built-in function spawn creates a new process.
- Each process has a process-id, pid.
  - The built-in function self() returns the pid of the calling process.
  - `spawn` returns the pid of the process that it creates.
  - The simplest form is `spawn(`*Fun*`)`.
    - ★ A new process is created – "the child".
    - ★ The pid of the new process is returned to the caller of `spawn`.
    - ★ The function *Fun* is invoked with no arguments in that process.
    - ★ The parent process and the child process are both running.
    - ★ When *Fun* returns, the child process terminates.

## Processes – a friendly example

```
hello(N)->
   [     spawn(fun() -> io:format(
           "hello world from process ~b~n", [I])
         end)
      || I <- lists:seq(1,N)
   ].
```

Running the code:

```
1> c(procs).
{ok,procs}
2> procs:hello(3).
hello world from process 1
hello world from process 2
hello world from process 3
[<0.40.0>,<0.41.0>,<0.42.0>]
```

## Messages

- To solve tasks in parallel, the processes need to communicate.
- Sending a message: `Pid ! Expr`.
  - `Expr` is evaluated, and the result is sent to process `Pid`.
  - We can send **any** Erlang term: integers, atoms, lists, tuples, . . .
- Receiving a message:

```
receive
    Pattern1 -> Expr1;
    Pattern2 -> Expr2;
    ...
    PatternN -> ExprN
end
```

  If there is a pending message for this process that matches one of the patterns,
  - The message is delivered, and the value of the `receive` expression is the value of the corresponding *Expr*.
  - Otherwise, the process blocks until such a message is received.
- Message passing is asynchronous: the sending process can continue its execution before the receiver gets the message.

## Adding two numbers using processes and messages

- The plan:
  - We'll spawn a process in the shell for adding two numbers.
  - This child process receives two numbers, computes the sum, and sends the result back to the parent.

```
add_proc(PPid) ->
   receive
      A -> receive
         B ->
            PPid ! A+B
      end
   end.

adder() ->
   MyPid = self(),
   spawn(fun() ->
      add_proc(MyPid)
   end).
```

```
3> Apid = procs:adder().
<0.44.0>
4> Apid ! 2.
2
5> Apid ! 3.
3
6> receive Sum -> Sum end.
5
```

## Reactive Processes and Tail Recursion

- Often, we want processes that do more than add two numbers together.
- We want processes that wait, receive a message, process the message, and then wait for the next message.
- In Erlang, we do this with recursive functions for the child process:

```
acc_proc(Tally) ->
   receive
     N when is_integer(N) ->
       acc_proc(Tally+N);
     {Pid, total} ->
       Pid ! Tally,
       acc_proc(Tally)
   end.

   accumulator() ->
     spawn(fun() ->
       acc_proc(0)
     end).
```

```
7> BPid = procs:accumulator().
<0.53.0>
8> BPid ! 1.
1
9> BPid ! 2.
2
10> BPid ! 3.
3
11> BPid ! {self(), total}.
{<0.33.0>, total}
12> receive T1 -> T1 end.
6
```

## Reactive Processes and Tail Recursion

- Often, we want processes that do more than add two numbers together.
- We want processes that wait, receive a message, process the message, and then wait for the next message.
- In Erlang, we do this with recursive functions for the child process:

```
acc_proc(Tally) ->
   receive
     N when is_integer(N) ->
       acc_proc(Tally+N);
     {Pid, total} ->
       Pid ! Tally,
       acc_proc(Tally)
   end.

   accumulator() ->
     spawn(fun() ->
       acc_proc(0)
     end).
```

```
13> BPid ! 4.
4
14> BPid ! {self(), total}.
{<0.33.0>, total}
15> BPid ! 5.
5
16> BPid ! 6.
6
17> BPid ! {self(), total}.
{<0.33.0>, total}
18> receive T2 -> T2 end.
10
19> receive T3 -> T3 end.
21
```

## Message Ordering

- Given two processes, *Proc1* and *Proc2*, messages sent from *Proc1* to *Proc2* are received at *Proc2* in the order in which they were sent.
- Message delivery is reliable: if a process doesn't terminate, any message sent to it will eventually be delivered.
- Other than that, Erlang makes no ordering guarantees.
  - In particular, the triangle inequality is not guaranteed.
  - For example, process *Proc1* can send message *M1* to process *Proc2* and after that send message *M2* to *Proc3*.
  - Process *Proc3* can receive the message *M2*, and then send message *M3* to process *Proc2*.
  - Process *Proc2* can receive messages *M1* and *M3* in either order.
  - Draw a picture to see why this is violates the spirit of the triangle inequality.

## Tagging Messages

- It’s a very good idea to include “tags” with messages.
- This prevents your process from receiving an unintended message:

  *“Oh, I forgot that another process was going to send me that. I thought it would happen later.”*

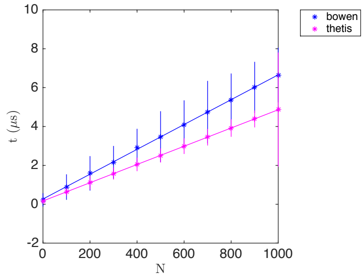
- For example, my accumulator might be better if instead of just receiving an integer, it received

```
{2, add}
```

## Timing Measurements

- We write parallel code to solve problems that would take too long on a single CPU.
- To understand performance trade-offs, I'll measure the time for some common operations in Erlang programs:
  - The time to make *N* recursive tail calls.
  - The time to spawn an Erlang process.
  - The time to send and receive messages:
    - Short messages.
    - Messages consisting of lists of varying lengths.

## Tail Call Time



`bowen.ugrad.cs.ubc.ca`:

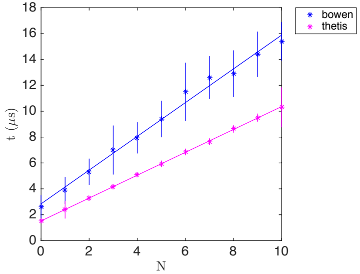
| | |
|---|---|
| $t = (6.4N + 269)\text{ns}$, | line of best fit |
| $t = 64.3\mu\text{s}$, | $N = 10K$ |
| $t = 640\mu\text{s}$, | $N = 100K$ |

`thetis.ugrad.cs.ubc.ca`:

| | |
|---|---|
| $t = (4.7N + 170)\text{ns}$, | line of best fit |
| $t = 46.9\mu\text{s}$, | $N = 10K$ |
| $t = 466\mu\text{s}$, | $N = 100K$ |

- Measurement: start the timing measurement, make *N* tail calls, end the timing measurement.
- The measurements on this slide and throughput the lecture were made using the `time_it:t` function from the course Erlang library.
  - `time_it:t(`*Fun* repeatedly calls *Fun* until about one second has elapsed. It then reports the average time and standard deviation.
  - `time_it:t` has lots of options.

## Process Spawning Time



bowen.ugrad.cs.ubc.ca:

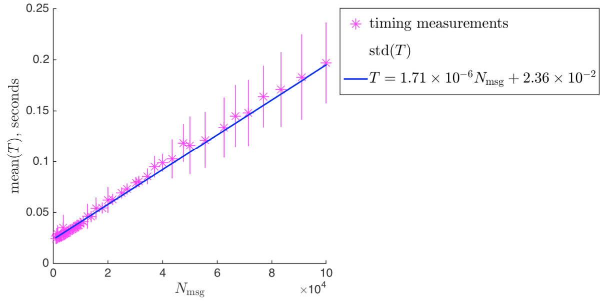
| | |
|---|---|
| $t = (1.30N + 2.8)\mu\text{s}$, | line of best fit |
| $t = 127\mu\text{s}$, | $N = 100$ |
| $t = 1.2\text{ms}$, | $N = 1000$ |

thetis.ugrad.cs.ubc.ca:

| | |
|---|---|
| $t = (0.88N + 1.5)\mu\text{s}$, | line of best fit |
| $t = 89.4\mu\text{s}$, | $N = 100$ |
| $t = 887\mu\text{s}$, | $N = 1000$ |

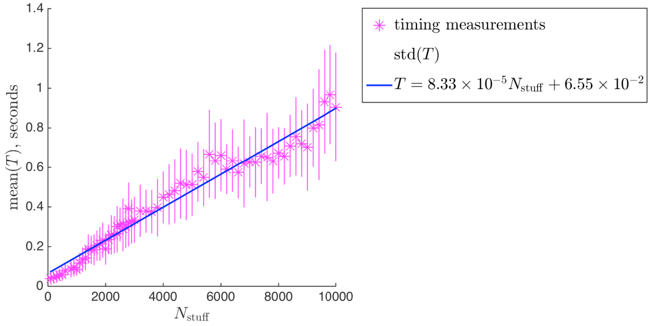
- Measurement: root spawns *Proc1*; *Proc1* spawns *Proc2*, and then *Proc1* exits; *Proc2* spawns *Proc3*, and then *Proc2* exits; ...; *ProcN* sends a message to the root process, and then *ProcN* exits. The root process measures the time from just before spawning *Proc1* until receiving the message from *ProcN*.

## Send+Receive Time



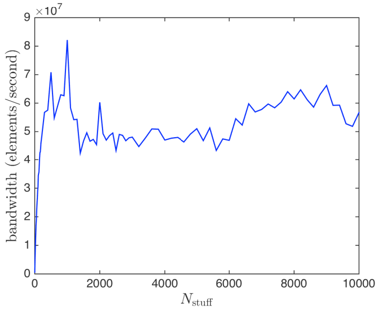
- Set-up: Two processes do a fixed amount of “work” while exchanging short messages with non-blocking receives.
- $N_{\text{msg}}$ is the number of messages sent and received by each process.
- The slope of the line is the time per message:
  - $\sim 1.7\mu\text{s}/\text{message}$ on `thetis.ugrad.cs.ubc.ca`, `erts` 18.2.
  - My laptop is about three-times faster. I’m running `erts` 19.2.

## Message Time vs. Message Size



- Set-up: as on the previous slide. This time each message consists of a list of $N_{\text{stuff}}$ small integers.
- Each process sends and receives 5000 messages per run.
- The slope of the line divided by 5000 is the time per element:
  - $\sim$17ns/message on `thetis.ugrad.cs.ubc.ca`, `erts` 18.2.

## Bandwidth vs. Message Size



Subtract the “non-message” time from the run-time and calculate:

$$\frac{N_{\text{msg}} \times N_{\text{stuff}}}{T}$$

To get elements per second.

- Bandwidth grows rapidly with message length for $N_{\text{stuff}} < 1000$, then drops.
  - Short messages have low bandwidth due to fixed overheads with each message.
  - I’m guessing that bandwidth drops some for messages with more than 1000 elements because the Erlang runtime is somehow optimized for short messages.

## Summarizing the numbers

- Interprocess operations such as `spawn`, `send`, and `receive` are **much** slower than operations within a single process such as + or a function call.
- An Erlang tail call is about 4.7ns, roughly 10 machine instructions.
- An Erlang tail call and add is about 4.7ns, roughly 10 machine instructions.
- Spawning a process is about 200$\times$ the cost of a tail call.
- For short messages, send and receive are about 350$\times$ the cost of a tail call.
  - The send/receive overhead can be amortized by sending longer message.
  - Each additional list element is about 3$\times$ the cost of a tail call.
  - Beware of any model that just counts the overhead and ignores the length, or just considers bandwidth and ignores the overhead.
- We will often refer to the ratio of the relationship between the time for interprocess operations and local operations as **big**.
  - In practice, **big** is 100 to 10000 for shared-memory computers.
  - **Big** can be even bigger for other architectures.

## How to Write Efficient Parallel Code

- Think about **communication costs**
  - Message passing is good – it makes communication explicit.
  - Pay attention to both the number of messages and their size.
  - Combining small messages into larger ones often helps.
- Think globally, but **compute locally**
  - Move the computation to the data, not the other way around.
  - Keep the data distributed across the parallel processes.
- Think about **big–*O***
  - If $N$ is the problem size, you want the computation time to grow faster with $N$ than the communication costs.
  - Then, your solution becomes more efficient for larger values of $N$.

## Summary

- Processes are easy to create in Erlang.
  - The `spawn` mechanism can be used to start other processors on the same CPU or on machines spread around the internet.
- Processes communicate through messages
  - Message passing is asynchronous.
  - The receiver can use patterns to select a desired message.
- Reactive processes are implemented with tail-recursive functions.
- Interprocess operations are much slower than local ones
  - This is a key consideration in designing parallel programs.
  - We'll learn **why** when we look at parallel architectures later this month.

## Preview

| | |
|---|---|
| **January 11: Reduce** | |
| Reading: | *Learn You Some Erlang*, Errors and Exceptions through A Short Visit to Common Data Structures |
| **January 13: Scan** | |
| Reading: | Lin & Snyder, chapter 5, pp. 112–125 |
| Mini-Assignment: | Mini-Assignment 2 due **10:00am** |
| **January 16: Generalized Reduce and Scan** | |
| Homework: | Homework 1 deadline for early-bird bonus (11:59pm) |
| | **Homework 2 goes out (due Feb. 1)** – Reduce and Scan |
| **January 18: Reduce and Scan Examples** | |
| Homework: | Homework 1 due **11:59pm** |

**January 20–27: Parallel Architecture**

**January 29–February 6: Parallel Performance**

**February 8–17: Parallel Sorting**

## Review Questions

- How do you spawn a new process in Erlang?
- What guarantees does Erlang provide (or not) for message ordering?
- Give an example of using patterns to select messages.
- Why is it important to use a tail-recursive function for a reactive process?
  - In other words, why is it a bad idea to use a head-recursive function for a reactive process.
  - The answer isn't explicitly on the slides, but you should be able to figure it out from what we've covered.
- Modify one of the examples in this lecture to use a time-out with one or more `receive` operations. Try it and show that it works.
- Implement the message flushing described in *LYSE* to show pending messages on a time-out. Demonstrate how it works.

## Supplementary material

- Debugging concurrent Erlang Code.
- Table of contents.

## Tracing Processes

When you implement a reactive process, it can be handy to trace the execution. Here's a simple approach:

- Add an `io:format` call when entering the function and after matching each receive pattern.
- Example:

```
acc_proc(Tally) ->
    io:format("~p:  acc_proc(~b)~n", [self(), Tally]),
    receive
        N when is_integer(N) ->
            io:format("~p:  received ~b~n", [self(), N]),
            acc_proc(Tally+N);
        Msg = {Pid, total}
            io:format("~p:  received ~p~n", [self(), Msg]),
            Pid ! Tally,
            acc_proc(Tally)
    end.
```

- Try it (e.g. with the example from slide 7.
- Don't forget to delete (or comment out) such debugging output before releasing your code.

## Time Outs

- If your process is waiting for a message that never arrives, e.g. because
  - You misspelled a tag for a message, or
  - The receive pattern is slightly different than the message that was sent, or
  - Something went wrong in the sending process, and it died before sending the message, or
  - You got the message ordering slightly wrong, and there's a cycle of processes waiting for each other to send something, or
  - . . .
- Then your process can wait forever, your Erlang shell can hang, and it's a very unhappy time in life.
- Time-outs can handle these problems more gracefully.
  - See Time Out in *LYSE*.
  - Note: time-outs are great for debugging. They should be used with great caution elsewhere because they are sensitive to changes in hardware, changes in the scale of the system, and so on.

## Table of Contents