Reduce

Mark Greenstreet

CpSc 418 – Jan. 13, 2016

Outline:

Objectives

- Introduce Erlang's features for concurrency and parallelism
  - Spawning processes.
  - Sending and receiving messages.
- The source code for the examples in this lecture is available here: procs.erl.

It's about time

- Time to make a tail-call: $\sim$ 5ns.
- Time to create a process: $\sim 1\mu$s.
- Time to send a small message: ping-pong: $\sim$ 360ns
- Time to send a small message: shuffle: 130 . . . 900ns
- Time to send a message vs. message size: TBD
- What does this say about writing parallel code?

The rest of this lecture

- Count3s.
- Count3s, brute-force.
- Count3s with a tree.
- The reduce pattern, and examples.

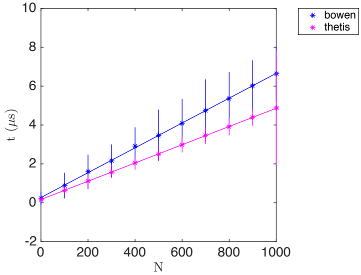
Preview

January 13: Reduce and Scan (simple)		
	Reading:	Lin & Snyder, chapter 5, pp. 112–125
	Mini-Assignment:	Mini-Assignment 2 due **10:00am**
January 15: Reduce and Scan (generalize)		
	Homework:	Homework 1 deadline for early-bird bonus (11:59pm)
January 18: Architecture Review		
	Reading:	Pacheco, Chapter 2, Sections 2.1 and 2.2.
	Homework:	Homework 1 due **11:59pm**
		Homework 2 goes out – parallel programming with Erlang
January 20: Shared-Memory Machines		
	Reading:	Pacheco, Chapter 2, Section 2.3
January 22: Distributed-Memory Machines		
	Reading:	Pacheco, Chapter 2, Sections 2.4 and 2.5.
January 25-29: Parallel Performance		
	Reading:	Pacheco, Chapter 2, Section 2.6.

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.

Tail Call Time



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

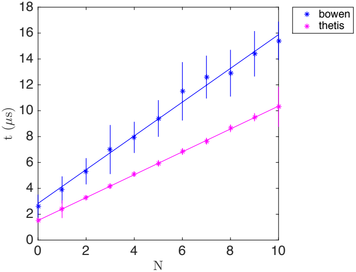
$t = (6.4N + 269)\mathrm{ns}$,	line of best fit
$t = 64.3\mu\mathrm{s}$,	$N = 10K$
$t = 640\mu\mathrm{s}$,	$N = 100K$

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

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

- 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`:

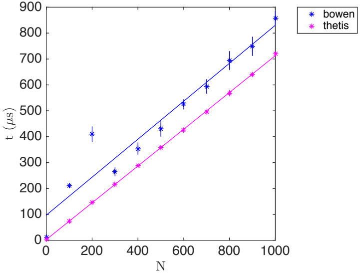
$t = (1.30N + 2.8)\mu\text{s}$,	line of best fit
$t = 127\mu\text{s}$,	$N = 100$
$t = 1.2\text{ms}$,	$N = 1000$

`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 *Proc3* 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*.

Ping-Pong Messages



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

$t = (0.73 + 97)\mu\text{s}$,	line of best fit
$t = 8.49\text{ms}$,	$N = 10K$
$t = 96\text{ms}$,	$N = 100K$

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

$t = (0.71 + 2.7)\mu\text{s}$,	line of best fit
$t = 7.17\text{ms}$,	$N = 10K$
$t = 72\text{ms}$,	$N = 100K$

- Measurement: root spawns two processes, *Ping* and *Pong*.
- In each of N rounds:
  - *Ping* sends a message to *Pong*.
  - *Pong* receives the message and then sends a message to *Ping*.
  - *Ping* receives the message from *Pong*.
- Two messages are sent and received per round.
  - The average time per message is about 360ns.

Shuffling Messages

- ping-pong can be “played” on a single CPU – only one process is active at a time.
- I wrote shuffle to try to keep many CPUs busy sending messages.
- With shuffle, we have P processors that have N rounds of messages. . . .
- Messages appear to have a sequential bottleneck.
  - Need to try again when the processes actually do something.

Shuffling on `bowen`

`bowen`

$N \backslash P$	4	8	16	32	64	128
1000	0.666	0.298	0.453	0.202	0.176	0.155
2000	0.448	0.322	0.438	0.202	0.172	0.153
3000	0.315	0.394	0.409	0.199	0.170	0.151
4000	0.316	0.428	0.273	0.198	0.164	0.152
5000	0.315	0.464	0.227	0.195	0.162	0.150
6000	0.319	0.504	0.231	0.201	0.163	0.150
7000	0.318	0.528	0.230	0.197	0.162	0.150
8000	0.317	0.560	0.259	0.195	0.161	0.151
9000	0.317	0.558	0.244	0.194	0.173	0.151
10000	0.315	0.560	0.231	0.194	0.175	0.154

Shuffling on `thetis`

`thetis`

$N \backslash P$	4	8	16	32	64	128
1000	0.413	0.338	0.303	0.208	0.155	0.104
2000	0.418	0.334	0.262	0.197	0.143	0.107
3000	0.425	0.333	0.260	0.179	0.124	0.130
4000	0.422	0.336	0.255	0.179	0.123	0.132
5000	0.428	0.340	0.260	0.178	0.130	0.130
6000	0.440	0.340	0.273	0.179	0.172	0.113
7000	0.430	0.328	0.276	0.188	0.298	0.112
8000	0.409	0.325	0.270	0.201	0.152	0.100
9000	0.403	0.408	0.274	0.193	0.153	0.128
10000	0.404	0.486	0.270	0.194	0.141	0.131