Reduce and Scan

Mark Greenstreet

CpSc 418 – Jan. 15, 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: $\sim$ 1μs.
- Time to send a small message (ping-pong): $\sim$ 360ns
  - Time to send a linked list of M small integers: $(15\mathrm{ns}) * M + 1.8\mu\mathrm{s}$ (on `bowen`)
- Time to send a small message: shuffle: $130 \ldots 900$ns
- Time to send a message vs. message size: TBD
- What does this say about writing parallel code?

Design Guidelines

- make trees, not chains:
  - When spawning processes, broadcasting a message, or collecing results, using a tree structure is generally better than have a master process spawn each worker, send a message from each worker, or receive a message from each worker.
- store data locally
  - It's tempting to store everything with the master process, send it out to workers to perform a task, and collect the results back.
    - This is a pattern that usually leads to parallel slow down.
    - E.g., on my laptop, parallel count 3s runs eight times slower than the sequential version if I use this approach.
  - Better that each worker keeps and uses its own data.
    - Example: large, distributed cloud file-systems and data analytics.
    - We'll do the same with count 3s by having each worker construct its onw random list of integers.

Count3s Design (version 1)

- A worker process has two parameters, `N` and `P`.
  - `N` is the total number of elements in the list.
    - The list will be distributed over the worker processes.
  - `P` is the total number of worker processes in this (sub-)tree.
- If a worker process is created with `P > 1`,
  - It creates two child processes.
  - Each is the root of a sub-tree with roughly `P/2` leaf processes.
  - Each of the two subtrees accounts for roughly `N/2` list elements.
  - This process waits to get the tallies from its children, adds them together, and sends them to its parent.
- If a worker process is created with `P == 1`,
  - It creates a list of `N` random integers.
  - It counts the 3s.
  - It sends its tally to its parent.

Count 3s Code (version 1, part 1)

```
worker(PPid, N, P) when P > 1 ->
   MyPid = self(),
   N2 = N div 2,
   P2 = P div 2,
   CPid1 = spawn(fun() -> worker(MyPid, N2, P2) end),
   CPid2 = spawn(fun() -> worker(MyPid, N-N2, P-P2) end),
   collect([CPid1, CPid2], ready),
   PPid !  MyPid, ready, ok,
   collect(PPid, go),
   CPid1 !  CPid2 !  MyPid, go, ok,
   [Tally1, Tally2] = collect([CPid1, CPid2], tally),
   PPid !  MyPid, tally, Tally1 + Tally2;

worker(PPid, N, 1) ->
   % see next slide
```

Count 3s Code (version 1, part 2)

```
worker(PPid, N, P) when P > 1 ->
   % see previous slide

worker(PPid, N, 1) ->
   MyPid = self(),
   Data = misc:rlist(N, 10),
   PPid ! {MyPid, ready, ok},
   collect(PPid, go),
   PPid ! {MyPid, tally, count3s(Data, 0)}.

collect(Pid, Tag) when is_pid(Pid) ->
   receive
      {Pid, Tag, Value} -> Value
      after 3000 ->
         flush_messages(),
         exit(time_out)
   end;
collect(PidList, Tag) when is_list(PidList) ->
   [collect(Pid, Tag) || Pid <- PidList].
```

Count3s Design: Critique Version 1

- To count 3s with `P` processes, we spawn a total of `P-1` processes:
  - `P` leaves to count the 3s in the sublists.
  - `P-1` non-leaf processes to combine results.
- Notice that the non-leaves just wait while the leaves are working.
  - This is wasteful.
  - Our design has more processes and more communication than needed.
- A better way:
  - When process `Pid1` spawns `Pid2`,
    - `Pid1` will assign its right sub-tree to `Pid2`.
    - `Pid1` will continue working on its left subtree.
  - Eventually, all of the processes are leaves, we go to work.

Count3s Design: The revised tree

This slide left blank so you can sketch the picture I'll draw on the board.

Count3s Design: Version 2

- Each process will keep a list of its children.
  - In more detail, each process will have two lists, one that goes from top-to-bottom in the tree, and the other will go from bottom to top.
  - Half of the processes are leaf-only – their lists of child processes will be empty.
- If process `Pid1` is created for a subtree with more than one node:
  - `Pid1` will spawn `Pid2` to handle the right subtree.
  - `Pid1` prepends `Pid2` to its bottom-to-top list of children.
  - `Pid1` continues wit the left subtree
- If process `Pid1` is created for a leaf node, it does the usual, count 3s work:
  - create a random list: `misc:rlist(N, 10)`
  - tell the parent it's ready: `ready(PPid, CPids_B2T)`
  - wait to receive a go: `go(PPid, CPids_T2B)`
  - count my own 3s: `count3s(Data)`
  - combine the results: `combine(PPid, CPids_B2T)`

Count3s Code: (version 2, part 1)

```
worker(PPid, N, P)
      when is_pid(PPid), is_integer(N), N >= 0, is_integer(P), P
   worker(PPid, N, P, []).

worker(PPid, N, P, CPidList) when P > 1 ->
   MyPid = self(),
   N2 = N div 2,
   P2 = P div 2,
   CPid = spawn(fun() -> worker(MyPid, N2, P2, []) end),
   worker(PPid, N-N2, P-P2, [CPid | CPidList]);

worker(PPid, N, 1, CPids_B2T) ->
   % At this point, CPidList is a list of all processes that we have
   % spawned from the bottom of the tree (a leaf) towards to top.
   CPids_T2B = lists:reverse(CPids_B2T),
   Data = misc:rlist(N, 10), % make our list
   ready(PPid, CPids_B2T),
   go(PPid, CPids_T2B),
   combine(PPid, CPids_B2T, count3s(Data)).
```

Count3s Code: (version 2, part 2)

```
ready(PPid, []) -> PPid ! self(), ready, ok;
ready(PPid, [CPid | CTail]) ->
   collect(CPid, ready),
   ready(PPid, CTail).

go(PPid, CPids) ->
   collect(PPid, go),
   go2(self(), CPids).

go2(_MyPid, []) -> ok;
go2(MyPid, [CPid | CTail]) ->
   CPid ! MyPid, go, ok,
   go2(MyPid, CTail).

combine(PPid, [], N3s) -> PPid ! self(), tally, N3s;
combine(PPid, [CPid | CTail], N3s) ->
   C3s = collect(CPid, tally),
   combine(PPid, CTail, N3s + C3s).
```

The Reduce Pattern

- It's a parallel version of *fold*, e.g. `lists:foldl` and `lists:foldr`.
- Reduce is described by three functions:

  Leaf(): What to do at the leaves, e.g.r `fun() -> count3s(Data) end`.
  Combine(): What to do at the root, e.g. `fun(Left, Right) -> Left+Right end`.
  Root(): What to do with the final result. For count 3s, this is just the identity function.

The `wtree` module

- Part of the course Erlang library.
- Operations on worker trees"

  `wtree:create(NProcs) -> [pid()].` Create a list of `NProcs` processes, organized as a tree.

  `wtree:broadcast(W, Task, Arg) -> ok.` Execute the function `Task` on each process in `W`. Note: `W` means "worker pool".

  `wtree:reduce(P, Leaf, Combine, Root) -> term().` A generalized reduce.

  `wtree:reduce(P, Leaf, Combine) -> term().` A generalized reduce where `Root` defaults to the identity function.

Store Locally

- As noted on slide 4, processes should store their data locally.
- How do we store data in a functional language?
  - Our processes are implemented as Erlang functions that receive messages, process the message, and make a tail-call to be ready to receive the next message.
  - We add a parameter to these functions, `State`, that is a mapping from *Keys* to *Values*.
- What this means when we write code:
  - Functions such as *Leaf* for `wtree:reduce` or *Task* for `wtree:broadcast` have a parameter for `State`.
  - `worker:put(State, Key, Value) -> NewState.` Create a new version of `State` that associates `Value` with `Key`.
  - `worker:get(State, Key, Default) -> Value.` Return the value associated with `Key` in `State`. If no such value is found, `Default` is returned. Note: `Default` can be a function in which case it is called to determine a default value – see the documentation.

Count3s using `wtree`

```
count3s_par(N, P) ->
   W = wtree:create(P),
   wtree:rlist(W, N, 10, 'Data'),
   wtree:barrier(W), % Need to add barrier to wtree
   wtree:reduce(W,
      fun(ProcState) -> count3s(workers:get(ProcState, 'Data'))
      fun(Left, Right) -> Left+Right end
   ).
```

Preview

January 18: Reduce and Scan (generalize)		
	Homework:	Homework 1 due **11:59pm**
		Homework 2 goes out – parallel programming with Erlang
January 20: Architecture Review		
	Reading:	Pacheco, Chapter 2, Sections 2.1 and 2.2.
January 22: Shared-Memory Machines		
	Reading:	Pacheco, Chapter 2, Section 2.3
January 25: Distributed-Memory Machines		
	Reading:	Pacheco, Chapter 2, Sections 2.4 and 2.5.
January 27: Parallel Performance: Speed-up		
	Reading:	Pacheco, Chapter 2, Section 2.6.
January 27: Parallel Performance: Overhead		

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.