Reduce

Mark Greenstreet

CpSc 418 – Jan. 11, 2017

Outline:

Objectives

- Understand why using a tree-structure for communication improves efficiency.
- Learn how to implement reduce using Erlang processes and messages.
- Learn how to use the `reduce` function in the course Erlang library.

Problem Statement

Given a list, *L* of *N* values, how can we use *P* processors to efficiently compute the sum of the values of the elements?

Possible in-class exercise:

- Divide class into groups of five or six.
- Hand each group a sheet of numbers. We could arrange the numbers is blocks, or perhaps hand each group a stack of five or six sheets, each of which has around 10 small integers (one to three digits) to add.
- Give them the task that the team that computes the sum of the numbers first wins (perhaps have a bag of M&M's or similar as a prize.
- See what they do.

Now, go back to the observations we made from the previous lecture.

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.
- Let's use a tail-call as the cost of an operation within a process.
- Spawning a process is about 200× the cost of a tail call.
- For short messages, send and receive are about 350× the cost of a tail call.
- For longer messages, the time grows with message length. Sending 100 numbers takes about twice as long as sending 1.

My guess is that many of the groups had a "team captain" who handed out the sheets of paper. Each team member would compute their local sum and report it to the team captain. The team captain computed the final sum and reported it (FTW).

We can do a bit of front-of-class theatre, you, Devon, me (and if we could get one or two others that would be great). Act it out with **slow** communication actions, e.g. "tai chi" style. Two problems should become apparent: starting where the team captain has all the data and distributes it is a bottleneck. Having everyone communicate send their result directly to the captain is a bottleneck.

How to Write Efficient Parallel Code

This is a review from the previous lecture.

- 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.

Interactive Exercise

- Design an efficient way to add N numbers using P processes.
- Should plan to start with each process having $\sim N/P$ values – this is the "Keep the data distributed across the parallel processes concept.
- Should "discover" the tree structure for communication
  - Point to bring out: we are not using a tree to get more parallelism in the final $P - 1$ additions. These adds don't take long enough to matter. The $P - 1$ communication actions **do** matter.
  - Reducing the depth of the communication actions from $P - 1$ (when the team captain handles all of them) to log P is what matters.

Now, translate it into code

- Let $N = 2^K$.
- We can have one process that creates a binary tree with N leaves.
- Each leaf process:
  - waits to receive a task with a tag.
  - does the task.
  - sends the result to its parent (with a parent provided tag).
- Each intermediate node:
  - Waits to receive a two functions and a tag.
  - Call the two functions *LeafTask* and *Combine*.
  - The node sends the two functions with a `left` or `right` tag to its children.
  - The node receives results from its children, combines them with *Combine* and sends the result with the parent provided tag to its parent.
  - We now discover that the leave probably receives

    {*LeafTask*, *Combine*, `LeftRightTag`, `PPid`}

    just like the intermediate nodes. The leaves just ignore the *Combine*.

Finish the code sketch

- The function for the process is pretty much like the intermediate nodes except
  - There's one function to create the tree.
  - Another function takes a process tree and the *LeafTask* and *CombineTask* functions and sends the result home.

It’s not quite that simple

- We want the leaf nodes to generate their arrays as one task, and compute the sums as another task.
- This means that the children need to “remember” state between the two tasks.
- There are a several possible solutions:
  - The *LeafTask* function could take an argument of `ProcState` and return a tuple of `{ValueForCombine, NewProcState}`.
  - The leaf process makes its recursive call with `NewProcState`.
  - Or, we could use the Erlang process dictionary with `erlang:put` and `erlang:get`, but that’s not very functional. I prefer the `ProcState` approach.
- And, we need to deal with end-of-life issues.
  - Use the atom `exit` instead of the {*LeafTask*, *Combine*} tuple.

We can do better

- **Note:** I'm not sure how far we can make it through this material with the various in-class activities. If we make it through the simple implementation of reduce on the previous slide, I'm happy. The rest of this is optional – equivalently, it's material we could move into the Jan. 13 or Jan. 16 lecture.
- With the design above, half of the processes sit around idle, while waiting for the leaves to do their work.
- We can make a tree where each process forwards messages to its right subtree(s) and then does its own *LeafTask*.
- I'm sure I've got a figure from some previous year, I'll find it.

But we don’t need to code the better version in class

- The better version is implemented in the course Erlang library.
- We now show how to do the reduce example with `wtree:reduce`.

Summary

Preview

After we make it through reduce, we'll cover

- scan – I want to have one lecture on scan by the end of the Jan. 16 lecture to have the students at a place that they can start on HW2.
- generalized scan and reduce.
- Then, we transition to $\sim$4 lectures on parallel architectures.

Review Questions