Erlang assignment 4

Parallel and distributed programming

Isak Samsten

Spring, 2023

Introduction

Advanced assignments for parallel and distributed Erlang using. This week has three assignments to test your knowledge of concurrent and distributed Erlang. The tasks are awarded a maximum of 20 points. The grades are distributed according to Table 1.

Table 1: Point to grade conversion table

Point	**Grade**
0 – 8	F
9 – 10	E
11 – 12	D
13 – 15	C
16 – 17	B
18 – 20	A

Submission

The following files should be submitted to iLearn:

Problem 1: `barrier.erl`

Problem 2: `allocator.erl`

Problem 3: `mapreduce.erl`

Problems

Problem 1 (6 points)

Re-write the barrier module we saw in Erlang Lecture 6 to not expect a fixed number of process to wait for. Instead, the barrier should wait until a specified set of processes have reached the barrier. *All other processes are allowed through the barrier without having to wait.* In Example 1.1, `barrier:wait/2` should block until all processes that use `[A, B]` (i.e., the references used in `barrier:start/1`) are waiting. The barrier should *not* wait for other processes. As a result, `do_a()` and `do_b()` are guaranteed to both have completed before `do_more(a)` and `do_more(b)`, whereas `do_more(c)` will run as soon as `do_c()` has completed.

Example 1.1

```
A = make_ref(), B = make_ref(), C = make_ref(),
Barrier = barrier:start([A, B]),
spawn(fun () -> do_a(), barrier:wait(Barrier, A), do_more(a) end),
spawn(fun () -> do_b(), barrier:wait(Barrier, B), do_more(b) end),
spawn(fun () -> do_c(), barrier:wait(Barrier, C), do_more(c) end).
```

More specifically, `barrier` should export `start/1` and `wait/2`, where `start(Refs)` should accept as argument a list, `Refs`, of references that should wait at the barrier. Next, `wait(Barrier, Ref)` should accept as the first argument the `Pid` returned by `barrier:start/1` and as the second argument a reference. If `Ref` is in `Refs`, `wait/2` should block until all references in `Refs` has arrived to `wait/2`. If `Ref` is not in `Refs`, the process is let through the barrier.

Problem 2 (7 points)

Re-write the resource *allocator* we saw in Erlang Lecture 6 to support named resources instead of an arbitrary number of resources.

More specifically, the `start/1`-function should accept a `map` of named resources `Pid = allocator:start(#{a=>10, b=>20, c=>30})`. Requesting resources is done through calling `request/2` with a list of resources to be requested e.g., `R = allocator:request(Pid, [a, c])`. The call to `request/2` should block until the resources are available and return a `map` with the resources, i.e., if any of the resources is unavailable the function waits until they have been released. Resources are released using `allocator:release(Pid, R)`, where `R` is a `map` of resources to be released. Note that you will not be able to guard your allocator-process using guard clauses when the allocator does not contain the requested resources. To keep the message in the allocators message queue one approach is to replicate the message by sending it to self, and thus keeping it in the message queue. One drawback of this is that the process will continue to process the message indefinitely as long as the resources have not been released, flooding the message queue and fully utilize the CPU. Your solution should not fully utilize the CPU, but instead the process should block until the requested resources have been released (however, remember that other processes might request resources that are still available and should be served).

The module shall export `start/1`, `request/2` and `release/2` as specified.

Problem 3 (7 points)

Use the implementation of `mapreduce.erl` we saw in Erlang Lecture 6 and extend it to:

1. minimize the workload of the master process by allowing mappers to send their data to the correct reducer (Problem 3.1)
2. Distribute mappers and reducers over multiple Erlang nodes (Problem 3.2)

Problem 3.1 (5 points)

In the `mapreduce`-module data is passed through the master-process from the mapper to the `reducers`. While this algorithm works well for small data, it limits the throughput for large data collections. Use the implementation `mapreduce.erl` we saw in Erlang Lecture 6 and extend it such that the `mapper` is responsible for sending its data to the correct `reducer`. Once a `mapper` has finished processing its data partition (and sent the result to the correct `reducers`), it notifies the master process. When the master process is notified that all `mappers` have completed, it notifies the `reduce`-processes that they can begin processing. Finally, the `reduce`-processes sends their result back to the master process which collects the results. The module must export the function `mapreduce/5`.

Problem 3.2 (2 points)

Extend the implementation such that it can spawn `mapper` and `reduce` processes on multiple distributed nodes[1]. Re-write, `mapreduce/5` to `mapreduce(Nodes, Mapper, Mappers, Reducer, Reducers, Input)`, where `Nodes` is a list of nodes to run `mappers` and `reducers` on. Write a test-case which uses `mapreduce/6` with **at least two Erlang nodes**. Export the functions `mapreduce/6` and `test_distributed/0`.

[1] On the same machine. See Erlang Lecture 7