DAT280 – Lab C: “Map-reduce in Erlang”

Olle Svensson (ollesv@student.chalmers.se)

Agazi Berihu (agazi@student.chalmers.se)

Solution 1: Distributing Map-Reduce

The solution builds on and is similar to *map_reduce_par*, some common code to the other solutions has been lifted out and put into *get_mapped* and *get_inputs*. Apart from that the we have added a node argument in *spawn_link* to specify on which node the process should be spawned. The processes are distributed between the available nodes in a round-robin manner by *spawn_mappers* and *spawn_reducers*.

Solution 2: Load-balancing Map-Reduce

We implemented a really basic worker pool where the main (master) process spawns the workers and assigns new work until no work is left. Once this is done it proceeds to collect all the results.

Solution 3: Fault-tolerant Map-Reduce

In this solution we extended the worker pool so that it now has a dedicated "pool" process for assigning work and handling crashing workers. The results are still sent directly to the "master" process. The

error handling is implemented by letting the "pool" process keep track of all current work in a map data structure. If a worker crashes it can do a lookup in the map to find the task that the worker was currently working on before it crashed. This task is then re-added to the list of tasks to be assigned.

We managed to create two nodes on one of our own computers, let one crash and still finish the *page_rank*-function. However we were unable to make this work on two separate lab computers. Either we didn't manage to set the modules up correctly or there is some flaw in or logic. Because of this we have no valid benchmark for this solution.

Benchmarks

The benchmarks where run on two lab computers (two physical nodes) with 4 logical cores each. We used 134MB of data.

Parallel: 16401.236ms.

Distributed: 10210.181ms.

Worker pool: 12000.609ms.

Fault tolerant: –

Conclusions

We noticed that worker pool was slower than the distributed version, despite only using as many worker as we have threads available. While the distributed created as many workers as there are chunks. We think that the reason the worker pool is a bit slower is because there is some idle time between when a worker has finished a task before it is assigned

a new task. This creates overhead which increases the runtime.

The distributed version does not have this problem and hence it is faster. However it is more naive and like mentioned in the PM would probably crash if used in a real world case.