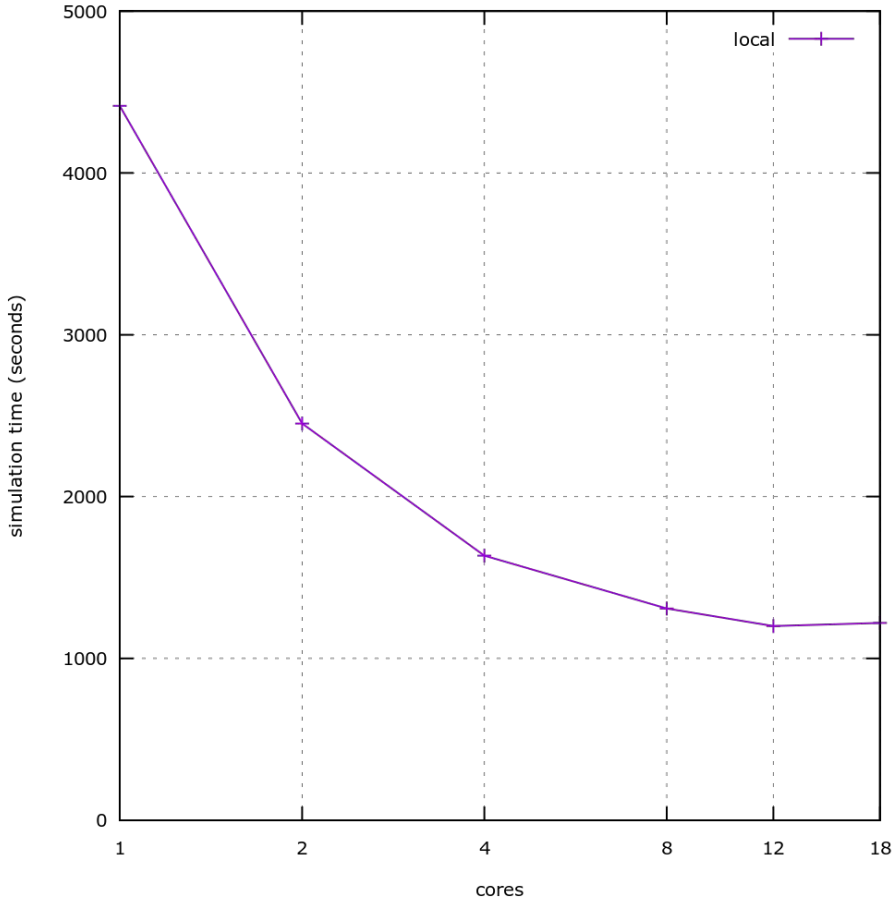
Parallelizing and distributing scientific software in Haskell

Francesco Mazzoli <francesco@fpcomplete.com>

The problem

- Need to execute a pure function, in parallel, on thousands of inputs, hundreds of times (say 200 rounds of 2000 executions).
- You have machines with 18+ cores at your disposal (thanks AWS!)
- You try `parMap`...


5000
4000
3000
2000
1000
0
simulation time (seconds)
1
2
4
8
12
18
cores
local

Why?

- As cores are added, the productivity goes down from 95% to 40%.
- When running in parallel, more Haskell objects are produced/discarded…
- Leading to more GC pauses that stop the world…
- Leading to a performance stall pretty soon (at around 10/12 cores).

How can we fix it?

Solution 1: make the pure function more GC-friendly

- For example: turn a lot of pure, sometimes boxed vectors into a single unboxed mutable state
- Net win -- faster program, more predictable operational behavior
- However, requires massive refactor of 80k lines of code codebase…
- ...for uncertain returns: we do not know if the new version would indeed be parallelizable.

Solution 2: use multiple Haskell runtimes

- The program gets slow because GC pauses stop *all* the threads.
- If each thread had its own GC, they'd all retain the high productivity!
- Also known as "do what Erlang does"
- Sadly, the only way to do this in Haskell is using several Haskell processes

Solution 2: use multiple Haskell runtimes

Reddit agrees!

> For high scalability, “fixing” GC is unlikely to work. With all the work put into it, the Java GC system is still horrible, while Erlang rules. Isolating heaps and using message passing is more scalable. I'd like to see more work in making multi-process (OS processes) programs work “fluently”.

“hastor” commenting on Measuring GC latencies in Haskell, OCaml, Racket

Multiple machines, too!

- The detailed approach gives you multi-machine parallelism, for free.
- Again, see Erlang: no distinction between message passing between threads and processes (possibly on different machines)

So how do we do it?

The problem in more detail

- We have `update :: state -> input -> (state, output)`
- We need to apply it in parallel to a `Vector state`, with a fixed input
- We need to do that repeatedly
  - Start with initial `Vector state` and the list of `input`
  - Update the `Vector state` by repeatedly applying the function with the `inputs`
  - End up with a `Vector (Vector output)`

Naive solution

- Spawn as many “worker” processes as you have cores
- Each worker process is a TCP server
- It accepts `(state, input)` messages, executes `update`, and replies with `(state, output)`
- Implement a `parMap` that uses the worker threads to parallelize the workload.

Naive solution: first problem (serialization)

- Since we're communicating with sockets, we need to serialize the state and the input.
- Serialization is super slow (`binary` and `cereal`)!
- `Doubles` are serialized in a very bizzarre way. Fixing that makes serialization 5x faster for us
- However, it's still not as fast as it could be because of some key design choices in `binary`/`cereal`
- Solution: write your own serialization library, store
- Serialization performance doubles, ¼ of memory consumption

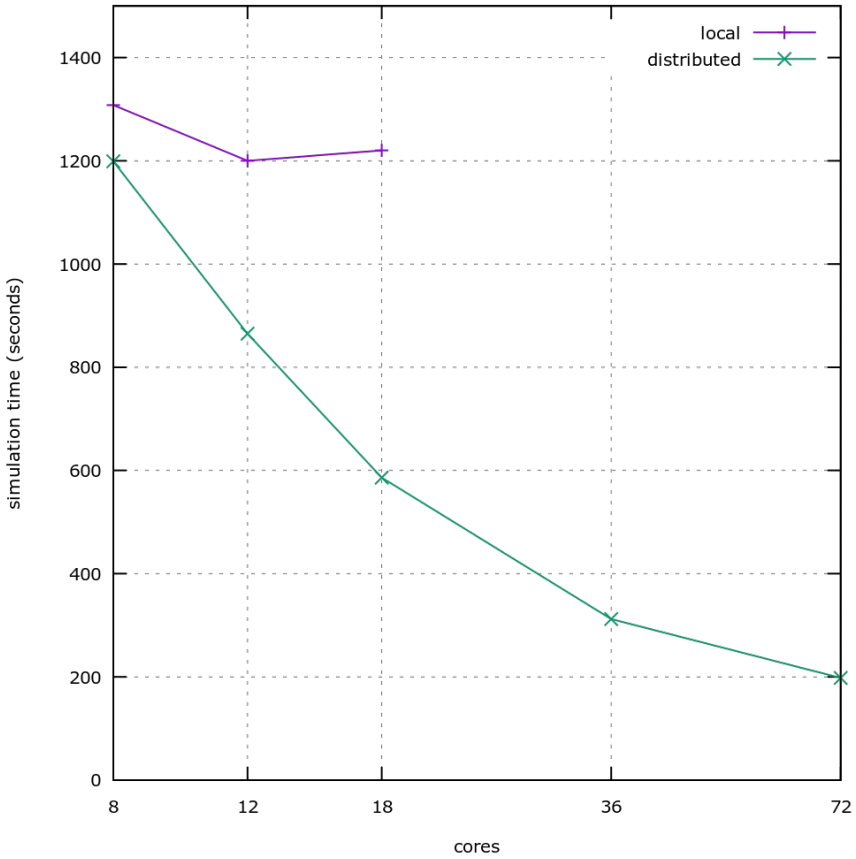
Naive solution: second problem (state size)

- In our application the states are big -- in the hundreds of KBs.
- This means that for each round of update, say with 1000 states, we'd have to serialize, transfer, and deserialize GBs of data.
- The time it takes to perform a single update is in the order of milliseconds
- Serializing and transferring the states ended up dominating the runtime (even with the faster serialization)

Solution: persist the states on the workers

- What we’re interested in is the intermediate outputs
- We can persist the states on the workers, and refer to them:
  - Distribute the states evenly to the workers at the beginning
  - Assign a unique id to each of them
  - Then, when needing to update the states, send just the input to the workers, and have the worker to return just the output
- Additional problem: we do not know which states are going to take longer to evolve
- Solution: evolve the states in batches, dynamically rebalance the work as we’re performing a round of updates

Results


local
distributed
1400
1200
1000
800
600
400
200
0
simulation time (seconds)
8
12
18
36
72
cores

Questions?

Sidebar 1: concurrent programming in Haskell is hard

- Haskell is probably the language with the most advanced concurrency primitives available
- However, I find that the community still hasn't settled on best practices to write correct concurrent code
- Problem 1: handling async exceptions is exceptionally hard. Partial solution: use our new safe-exceptions package.
- Problem 2: handling threads correctly is exceptionsally hard. Solution: *never* use `forkIO` always use a recent version of `async`

Sidebar 2: the Haskell RTS is sometimes surprising

- Deadlock detection is very, *very* subtle. For example:
  - If you're waiting on a deadlocked thread the waiting thread will be killed.
  - The only workaround for this is to blindly swallow all deadlock exceptions thrown at the blocked thread.
  - Possible solution in the future: tag deadlock exceptions with a trail of the blockage, so that we can see the thread at the root of the problem.
  - See this issue and related ones for examples of the subtleness.

Sidebar 2: the Haskell RTS is sometimes surprising

- Haskell will interrupt syscalls in foreign code every few milliseconds.
  - GHC sends `SIGVTALRM` at a fixed interval, see the documentation for `-V`
  - If a syscall is executed
  - Example: a `fork` in foreign code could never terminate because it took longer than 25ms (the default RTS clock), leading to an unkillable process.