# Parallel Performance Tuning for Haskell: An Example

Andrés Sicard-Ramírez

Ciclo de Conferencias Apolo
Universidad EAFIT
2017-11-08

# Motivation

- Haskell is a pure functional programming.
- Tuning (Haskell) parallel programs is not a trivial task:
  - garbage collection
  - lazy evaluation
  - task granularity
  - data dependencies
  - speculation
  - etc.

# A Compiler for Haskell

### GHC

The Glasgow Haskell Compiler for Haskell.

### GHC run-time system

*"To make an executable program, the GHC system compiles your code and then links it with a non-trivial runtime system (RTS), which handles storage management, thread scheduling, profiling, and so on."* (from GHC 8.2.1 user manual)

# Parallel Computing in Haskell

- Parallel programming in Haskell is deterministic
  - If a parallel program gives a result it always is the same.
  - *"Deterministic parallel programming is the best of both worlds: testing, debugging and reasoning can be performed on the sequential program, but the program runs faster when processors are added."* *

*Marlow, S. (2012). Parallel and Concurrent Programming in Haskell, p. 342.

# Parallel Computing in Haskell

- Parallel Haskell programs do not explicitly deal with synchronisation or communication
  - *"Synchronisation is the act of waiting for other tasks to complete, perhaps due to data dependencies. Communication involves the transmission of results between tasks running on different processors. Synchronisation is handled automatically by the GHC runtime system and/or the parallelism libraries. Communication is implicit in GHC since all tasks share the same heap, and can share objects without restriction."* [*]
  - *"This is both a blessing and a curse."* [†]

[*] Marlow, S. (2012). Parallel and Concurrent Programming in Haskell, p. 343.
[†] Marlow, S. (2013). Parallel and Concurrent Programming in Haskell, p. 6.

# Sparks

GHC run-time system creates a sparks for some expressions. *"Sparks may be evaluated at some point in the future, or they might not—it all depends on whether there is a spare core available."* *

*Marlow, S. (2013). Parallel and Concurrent Programming in Haskell, p. 25.

# Sparks

During the program execution a spark can be:

- **converted** the spark was executed
- **overflowed** the spark pool is full and the spark was dropped
- **dud** the sparked expression is already evaluated
- **GC'd** the sparked expression was found to be unused by the program
- **fizzled** the expression was unevaluated at the time it was sparked but was later evaluated independently by the program

# Two Computations

The Fibonacci and the Ackermann functions:

$$\mathrm{fib} : \mathbb{N} \to \mathbb{N}$$

$$\mathrm{fib}\,(n) = \begin{cases} 0, & \text{if } n = 0; \\ 1, & \text{if } n = 1; \\ \mathrm{fib}\,(n-1) + \mathrm{fib}\,(n-2), & \text{otherwise}; \end{cases}$$

$$\mathrm{ack} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$

$$\mathrm{ack}\,(x, y) = \begin{cases} y + 1, & \text{if } x = 0; \\ \mathrm{ack}\,(x-1, 1), & \text{if } y = 0; \\ \mathrm{ack}\,(x-1, \mathrm{ack}\,(x, y-1)), & \text{otherwise.} \end{cases}$$

# Running Example

### Task

To compute

$$\mathrm{fib}\,(39) + \mathrm{ack}\,(3, 11).$$

### Reference and versions

The following examples were adapted from [Jones, Marlow and Singh 2009] and they were tested with GHC 8.2.1, the parallel library 3.2.1.1 and ThreadScope 0.2.9.

# Example 1: Sequential Implementation

## Source code

See `Example1.hs`.

# Example 1: Sequential Implementation

## Source code

See `Example1.hs`.

## Running `Example1.hs`

```
$ ghc Example1.hs
$ ./Example1
63262367
```

# Basic Parallelism: The par Function

Basic parallelism is supported by the functions*

```
par  :: a → b → b
pseq :: a → b → b
```

from the parallel library, where

- `par a b` is semantically equivalent to `b` and
- `par` creates a spark for its first argument.

*Marlow, S., Peyton Jones, S. and Singh, S. (2009). Runtime Support for Multicore Haskell.

# Example 2: A Wrong Parallelisation

## Description

Using the `par` function for running on parallel the computations in Example 1.

# Example 2: A Wrong Parallelisation

## Description

Using the `par` function for running on parallel the computations in Example 1.

## Source code

See `Example2.hs`.

# Example 2: A Wrong Parallelisation

## Description

Using the `par` function for running on parallel the computations in Example 1.

## Source code

See `Example2.hs`.

## Running `Example2.hs`

```
$ ghc -threaded Example2.hs
$ ./Example2 +RTS -N2
63262367
```

# ThreadScope

## Description

ThreadScope is a graphical tool for performance profiling of parallel Haskell programs.

# ThreadScope

## Description

ThreadScope is a graphical tool for performance profiling of parallel Haskell programs.

## Installation

```
$ cabal update
$ cabal install threadscope
```

## Compiling your program

```
$ ghc -threaded -eventlog Foo.hs
```

## Running your program

```
$ ./Foo +RTS -N2 -l
```

## Viewing the eventlog

```
$ threadscope Foo.eventlog
```

# Example 2: A Wrong Parallelisation

From the ThreadScope output we know that we are only using one processor.

# Using RTS Statistics

## The `+RTS -s -RTS` option

This option produces run-time system statistics including sparks information:

| | |
|---|---|
| MUT time | the time running the program |
| GC time | the time spent performing garbage collection |
| Total time | MUT time + GC time + ... |
| Wall clock time | elapsed time |

# Example 2: A Wrong Parallelisation

## What it is wrong?

```
SPARKS: 1 (0 converted, 0 overflowed, 0 dud, 0 GC'd, 1 fizzled)

INIT    time    0.000s  (  0.002s elapsed)
MUT     time   19.948s  ( 23.197s elapsed)
GC      time   10.092s  (  5.561s elapsed)
EXIT    time    0.000s  (  0.002s elapsed)
Total   time   30.040s  ( 28.761s elapsed)
```

# Example 3: Maybe a Lucky Parallelisation

## Description

Swapping the computation of `fib` and `ack` in Example 2.

## Source code

See `Example3.hs`.

# Example 3: Maybe a Lucky Parallelisation

## RTS statistics

```
SPARKS: 1 (1 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)

INIT    time    0.000s  (  0.002s elapsed)
MUT     time   12.472s  ( 11.831s elapsed)
GC      time   15.988s  (  5.550s elapsed)
EXIT    time    0.004s  (  0.009s elapsed)
Total   time   28.464s  ( 17.391s elapsed)
```

## Feature of a good parallelisation

*“A profitably parallel program will have a wall clock time (elapsed time) which is less than the total time.”* *

*Jones, D., Marlow, S. and Singh, S. (2009). Parallel Performance Tuning for Haskell, p. 82.

# Example 3: Maybe a Lucky Parallelisation

## What it is wrong?

The fix works by accident because GHC could use a different order of evaluation for `(+)`.

# Basic Parallelism: The pseq Function

The semantics of the function

```
pseq :: a → b → b
```

is given by

$$\texttt{pseq a b} = \begin{cases} \bot, & \text{if } \texttt{a} = \bot; \\ \texttt{b}, & \text{otherwise.} \end{cases}$$

that is, `pseq a b`, evaluates `a` before `b` and returns the value of `b`.*

*Marlow, S., Peyton Jones, S. and Singh, S. (2009). Runtime Support for Multicore Haskell.

# Example 4: A Correct Parallelisation

## Description

Using the `par` and `pseq` functions for running on parallel the computations.

## Source code

See `Example4.hs`.

# Example 4: A Correct Parallelisation

## Description

Using the `par` and `pseq` functions for running on parallel the computations.

## Source code

See `Example4.hs`.

## RTS statistics

```
SPARKS: 1 (1 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)

INIT    time    0.000s  (  0.000s elapsed)
MUT     time   12.004s  ( 11.252s elapsed)
GC      time   15.168s  (  5.471s elapsed)
EXIT    time    0.000s  (  0.007s elapsed)
Total   time   27.172s  ( 16.730s elapsed)
```

# Was Successful the Parallelisation?

## Feature of a good parallelisation

*"A profitably parallel program will have a wall clock time (elapsed time) which is less than the total time.*

Footnote: *"although to measure actual parallel speedup, the wall-clock time for the parallel execution should be compared to the wall-clock time for the sequential execution."* *

*Jones, D., Marlow, S. and Singh, S. (2009). Parallel Performance Tuning for Haskell, p. 82.

# Was Successful the Parallelisation?

## Feature of a good parallelisation

*"A profitably parallel program will have a wall clock time (elapsed time) which is less than the total time.*

Footnote: *"although to measure actual parallel speedup, the wall-clock time for the parallel execution should be compared to the wall-clock time for the sequential execution."* *

## Wall-clock time for Example 1 (Sequential Implementation) and Example 4 (A Correct Parallelisation)

```
Example 1: Total   time   23.652s  ( 23.749s elapsed)
Example 4: Total   time   27.172s  ( 16.730s elapsed)
```

*Jones, D., Marlow, S. and Singh, S. (2009). Parallel Performance Tuning for Haskell, p. 82.

# References


Jones, D., Marlow, S. and Singh, S. (2009). Parallel Performance Tuning for Haskell. In: Proceedings of the ACM SIGPLAN 2009 Haskell Workshop, pp. 81–92. DOI: 10.1145/1596638.1596649.

Marlow, S. (2012). Parallel and Concurrent Programming in Haskell. In: Central European Functional Programming School (CEFP 2011). Ed. by Zsók, V., Horváth, Z. and Plasmeijer, R. Vol. 7241. Lecture Notes in Computer Science, pp. 333–401. DOI: 10.1007/978-3-642-32096-5_7.

— (2013). Parallel and Concurrent Programming in Haskell. O'Reilly Media, Inc.

Marlow, S., Peyton Jones, S. and Singh, S. (2009). Runtime Support for Multicore Haskell. In: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP '09), pp. 65–77. DOI: https://doi.org/10.1145/1631687.1596563.