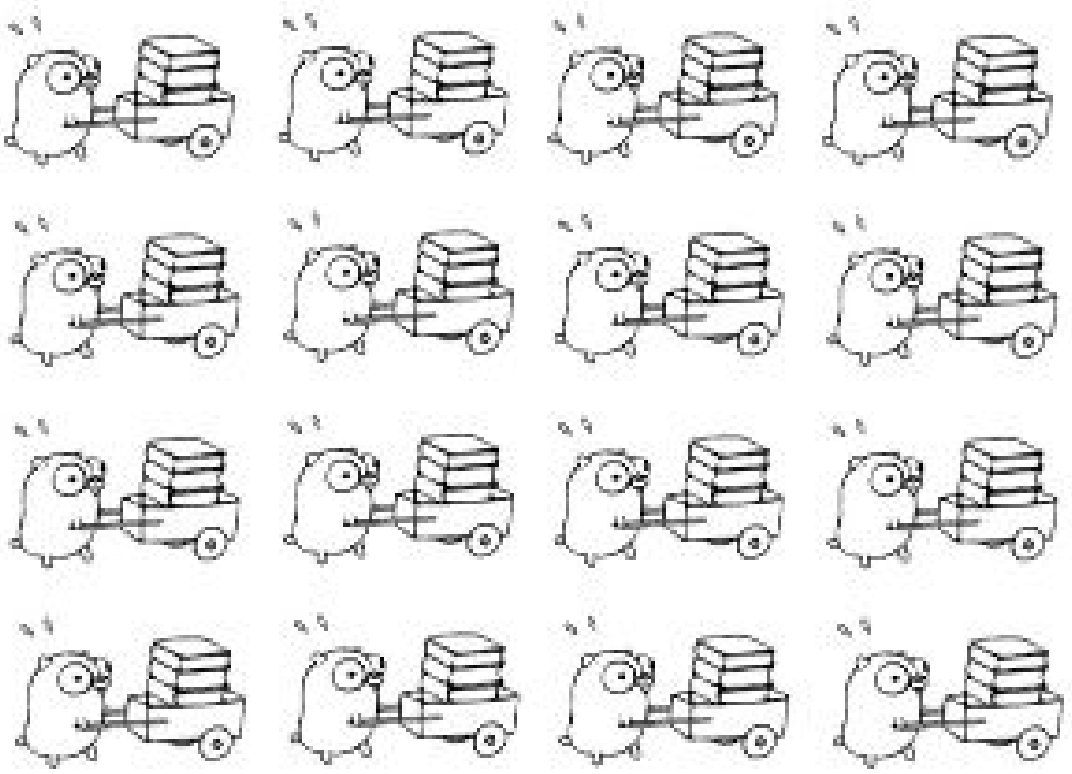
Tutorial: The Par Monad -Simplifying Parallel Haskell Programming

A quick tutorial for anyone who want´s to up their Haskell game.

Created by Mohamed Hassainia and Miki Swahn,
As part of the course Parallel Functional Programming (DAT280) at Chalmers University of Technology 2018.



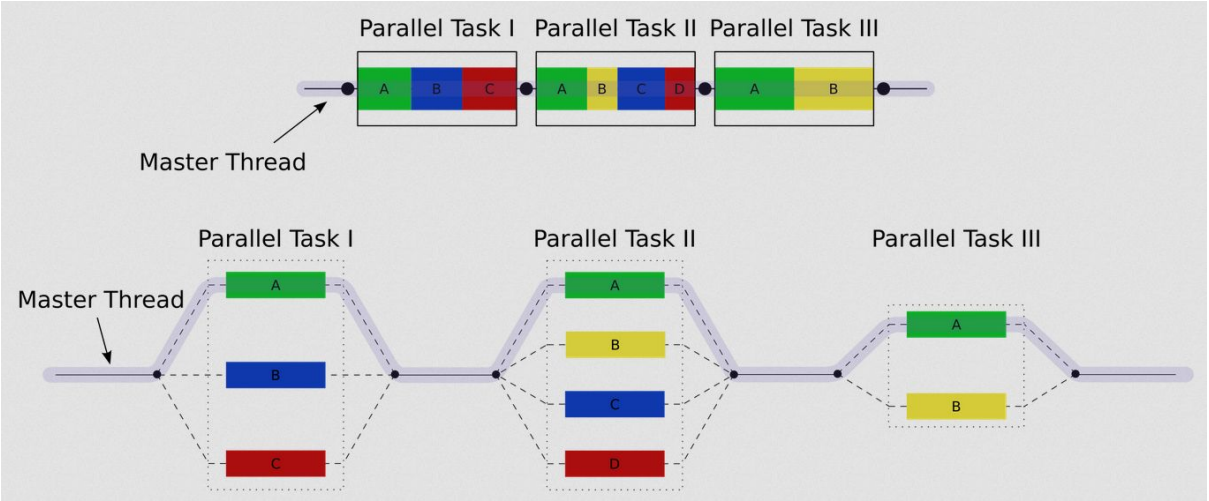
Parellel work is faster than sequential. Image source: Intel,
https://software.intel.com/en-us/blogs/2013/06/18/go-parallel

Introduction

The Par monad is a Haskell library for parallelizing computations. The execution is deterministic as long as the computations is pure, as opposed to I/O computations. But good news are, I/O computations can also be parallelized with the Par monad, only they won't be determinitisic for natural reasons. The Par monad has a very simple interface which abstracts away the parallel backend, making it easy to use!

Why parallelize?

First of all, why even bother? What are the actual benefits? Efficiency, is the short answer. Modern computers have multiple cores and multiple threads enabling them do do several computations at the same time. An unused clock cycle cannot be saved and used later, it is a time-bound capacity. Therefore, in order to be as efficient as possible, computations have to be done in parallel.



The difference between a sequential program and a parallel one.You just have to compare Parallel task 1 on the threads: in the sequential case A is done first, while B waits to be executed next, and C is not executed until both A and B have been completed. In the parallel case A, B and C are all done simultaneously. Image source: DaTuOpinion.com, *http://www.datuopinion.com/openmp*

There are various ways of achieving this. With speculative evaluation modern computers already parallelize our computations behind the scenes, without us having to tell them to. Additionally, you as a programmer can specify in your code that certain computations shall be done in parallel. By tapping into this opportunity your code can become much faster, since your computer's scheduler will not figure out all the things it can parallelize without your help.

So should you parallelize everything? No. The truth is. small computations take longer time if executed in parallel, because merely initializing new processes consume resources. So when using the Par monad, you should test your code to see that the computations you run in parallel are big enough to benefit from it. A good advice in to use the benchmarking tool *Criterion* which is accessible for free. The Par monad gives you a simple API which makes it easy for you to parallelize your code. Now, you can experiment and create really fast programs!

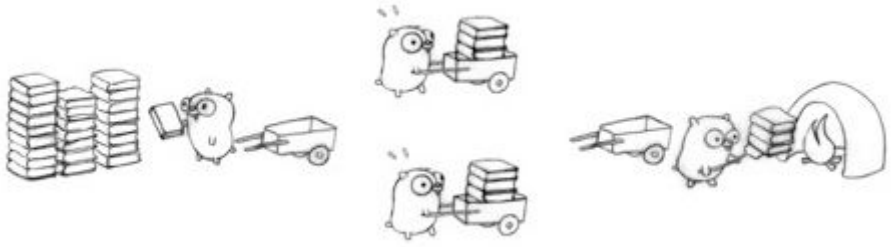
How do I use the Par monad?

Okay, there are six lines of boileplate code you should learn. (How simple!) Here it goes:

```
runPar $ do
    fx <- spawnP (f x)    -- start evaluating (f x)
    gx <- spawnP (g x)  -- start evaluating (g x)
    a  <- get fx             -- wait for fx
    b  <- get gx             -- wait for gx
    return (a,b)             -- return results
```

Code source: Hackage, http://hackage.haskell.org/package/monad-par-0.3.4.8/docs/Control-Monad-Par.html

Also, don't forget to "import Control.Monad.Par". What the above code does, is computing the values of fx and gx in parallel, which hopefully should be quicker than evaluating them sequentially. The synchronization is accomplished with the "get-calls" that wait for the parallel computations to be performed and then assigns the values.



To speed up tings, fx and gx are being computed by separate processes, as illustrated by the two hamsters that separately carry a load of bricks to the fire. Image source: Intel, *https://software.intel.com/en-us/blogs/2013/06/18/go-parallel* altered.

What does the Par monad do?

The core concept of parallelism, is dividing a problem into independent bits that can be computed simultaneously and then combining them to the total solution. From here, two challenges arises: problem decomposition and process interaction. Problem decomposition is the task of figuring out which parts of a computational problem that can be computed separately. These subproblems can of course be dependent on each other, which requires a clever way of gluing them together. Process interaction is the issue of making these individual computations synchronize in some way, which is required since they produce a joint result.

Now, the Par monad resolves the process interaction with the “get” call, which ensures the main process waits for the subprocesses to finish before proceeding with the sequential computations. The Problem decomposition is what you as a programmer have to figure out. It’s how you divide the whole problem into function calls that can be spawned separately. Thanks to the Par monad, parallel programming in Haskell does not have to be more complicated than this. The details of how processes are spawned and such is none of your concern. Instead, you can focus on the algorithm!