Under consideration for publication in J. Functional Programming

Arrows for Parallel Computations

MARTIN BRAUN
University Bayreuth, 95440 Bayreuth, Germany
PHIL TRINDER
Glasgow University, Glasgow G12 8QQ, Scotland
and OLEG LOBACHEV
University Bayreuth, 95440 Bayreuth, Germany

Abstract

Arrows form**OL: are?** a general interface for computation and pose therefore as an alternative to monads for API design. We express parallelism using this concept. This is a new way to represent parallel computation. We define an Arrows-based interface for parallelism and implement it using multiple parallel Haskells. **OL: Benefits:** In this manner we are able to bridge across various parallel Haskells with a common interface.

This new way of writing parallel programs has the benefit of being portable across flavours of parallel Haskells.**OL: Wdh?** Each parallel computation is an arrow, they can be composed and transformed as such. We thus introduce some syntactic sugar to provide parallelism-aware arrow combinators.

We also define several parallel skeletons with our framework. Benchmarks shows that our framework does not induce too much overhead performance-wise.

Contents

Contents

1 Introduction

OL: todo, reuse 5.5, "Impact" at the end and more

blablabla arrows, parallel, haskell.

Contribution HIT HERE REALLY STRONG

Structure The remaining text is structures as follows. Section 2 briefly introduces known parallel Haskell flavours and gives an overview of Arrows to the reader (Sec. **??**). Section **??** discusses related work. Section **??** defines Parallel Arrows and presents a basic interface. Section **??** defines Futures for Parallel Arrows, this concept enables better communication. Section **??** presents some basic algorithmic skeletons (parallel `map` with and without load balancing, `map-reduce`) in our newly defined dialect. More advanced ones are showcased in Section **??** (`pipe`, `ring`, `torus`). Section **??** shows the benchmark results. Section **??** discusses future work and concludes.

2 Background

2.1 Short introduction to parallel Haskells

There are already several ways to write parallel programs in Haskell. As we will base our parallel arrows on existing parallel Haskells, we will now give a short introduction to the ones we use as backends in this paper.

In its purest form, parallel computation (of functions) can be looked at as the execution of some functions `a -> b` in parallel, as also Figure **??** symbolically shows:

$$parEvalN :: [a \to b] \to [a] \to [b]$$

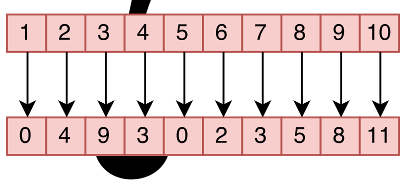

Figure 1: Schematic illustration of `parEvalN`.

Before we go into detail on how we can use this idea of parallelism for parallel Arrows, as a short introduction to parallelism in Haskell we will now implement `parEvalN` with several different parallel Haskells.

2.1.1 Multicore Haskell

Multicore Haskell (**??**) is way to do parallel processing found in standard GHC.[1] It ships with parallel evaluation strategies (**??**) for several types which can be applied with `using :: a -> Strategy a -> a`. For `parEvalN` this means that we can just apply the list of functions `[a -> b]` to the list of inputs `[a]` by zipping them with the application operator `$`. We then evaluate this lazy list `[b]` according to a `Strategy [b]` with the `using :: a -> Strategy a -> a` operator. We construct this strategy with `parList :: Strategy a -> Strategy [a]` and `rdeepseq :: NFData a => Strategy a` where the latter is a strategy which evalutes to normal form. To ensure that programs that use `parEvalN` have the correct evaluation order, we annotate the computation with `pseq :: a -> b -> b` which forces the compiler to not reorder multiple `parEvalN` computations. This is particularly necessary in circular communication topologies like in the `torus` or `ring` skeleton that we will see in chapter **??** which resulted in deadlock scenarios when executed without `pseq` during testing for this paper.

[1] Multicore Haskell on Hackage is available under `https://hackage.haskell.org/package/parallel-3.2.1.0`, compiler support is integrated in the stock GHC.

$$
\begin{array}{l}
\mathit{parEvalN} :: (\mathit{NFData}\ b) \Rightarrow [a \to b] \to [a] \to [b] \\
\mathit{parEvalN}\ \mathit{fs}\ \mathit{as} = \textbf{let}\ \mathit{bs} = \mathit{zipWith}\ (\$)\ \mathit{fs}\ \mathit{as} \\
\quad \textbf{in}\ (\mathit{bs}\ \text{`}\mathit{using}\text{`}\ \mathit{parList}\ \mathit{rdeepseq})\ \text{`}\mathit{pseq}\text{`}\ \mathit{bs}
\end{array}
$$

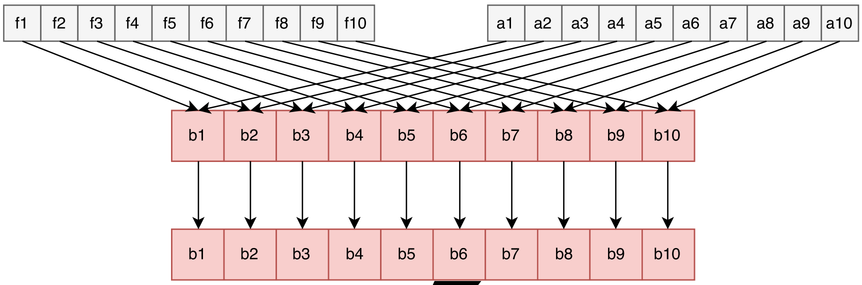

Figure 2: Dataflow of the Multicore Haskell parEvalN version

OL: Evaluation step explicitly shown?

2.1.2 ParMonad

The `Par` monad[2] introduced by **?**, is a m[illegible]esigned for composition of parallel programs.

Our parallel evaluation function `parEvalN` can be defined by zipping the list of `[a -> b]` with the list of inputs `[a]` with the application operator $ just like with Multicore Haskell. Then, we map over this not yet evaluated lazy list of results `[b]` with `spawnP :: NFData a => a -> Par (IVar a)` to transform them to a list of not yet evaluated forked away computations `[Par (IVar b)]`, which we convert to `Par [IVar b]` with `sequenceA`. We wait for the computations to finish by mapping over the `IVar b`'s inside the `Par` monad with `get`. This results in `Par [b]`. We finally execute this process with `runPar` to finally get `[b]` again.

explain problems with laziness here. Problems with torus

$$
\begin{array}{l}
\mathit{parEvalN} :: (\mathit{NFData}\ b) \Rightarrow [a \to b] \to [a] \to [b] \\
\mathit{parEvalN}\ \mathit{fs}\ \mathit{as} = \mathit{runPar}\ \$ \\
\quad (\mathit{sequenceA}\ \$\ \mathit{map}\ (\mathit{spawnP})\ \$\ \mathit{zipWith}\ (\$)\ \mathit{fs}\ \mathit{as}) \gg\!= \mathit{mapM}\ \mathit{get}
\end{array}
$$

2.1.3 Eden

Eden (**??**) is a parallel Haskell for distributed memory and comes with a MPI and a PVM backends.[3] This means that it works on clusters as well so it makes sense to have a Eden-based backend for our new parallel Haskell flavour.

[2] It can be found in the `monad-par` package on hackage under `https://hackage.haskell.org/package/monad-par-0.3.4.8/`.

[3] See also `http://www.mathematik.uni-marburg.de/~eden/` and `https://hackage.haskell.org/package/edenmodules-1.2.0.0/`.

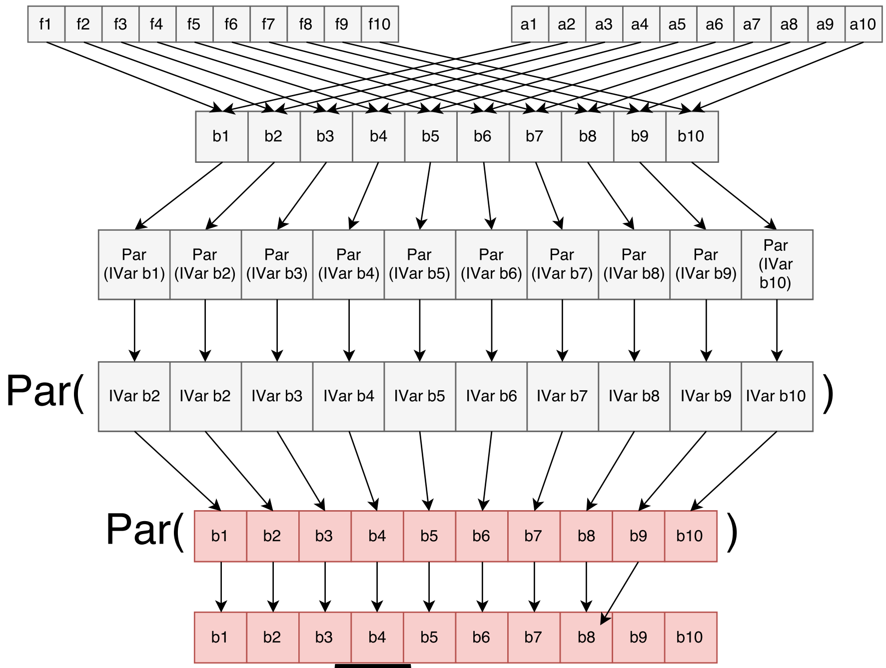

Figure 3: Dataflow [redacted] the Par Monad parEvalN version

Eden was designed to work on clusters, but with a further simple backend it operates on multicores. However, in contrast to many other parallel Haskells, in Eden each process has its own heap. This seems to be a waste of memory, but with distributed programming paradigm and individual GC per process, Eden yields good performance results also on multicores (**??**).

While Eden also comes with a monad `PA` for parallel evaluation, it also ships with a completely functional interface that includes a `spawnF` function that allows us to define `parEvalN` directly:

$$parEvalN :: (Trans\ a, Trans\ b) \Rightarrow [a \to b] \to [a] \to [b]$$
$$parEvalN = spawnF$$

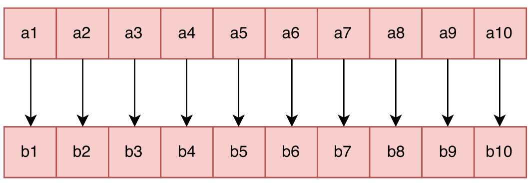

Figure 4: Dataflow of the Eden parEvalN version