Under consideration for publication in J. Functional Programming

Arrows for Parallel Computation

MARTIN BRAUN
University Bayreuth, 95440 Bayreuth, Germany
OLEG [illegible]CHEV
University Bay[illegible]uth, 95440 [illegible]uth, Germany
a[illegible]PHIL TRIN[illegible]
Glasgow Univ[illegible] Glasgow, G[illegible]QQ, Scotland

Abstra[illegible]

Arrows are a general interface for computation and a[illegible]lternative to Monads for API design. In contrast to Monad-based parallelism, we explore the use o[illegible] Arrows for specifying generalised parallelism. Specifically, we define an Arrow-based language a[illegible]d implement it using multiple parallel Haskells.

As each parallel computation is an Arrow, such parallel Arrows (PArrows) can be readily composed and transformed as such. To allow for more sophisticated communication schemes between computation nodes in distributed systems, we utilise the [illegible]pt of Futures to wrap direct communication.

To show that PArrows have similar expressiv[illegible]r as existing parallel languages, we implement several algorithmic skeletons and four benchmarks. Benchmarks show that our framework does not induce any notable performance overhead. We conclude that Arrows have considerable potential for composing parallel programs and for producing programs that can execute on multiple parallel language implementations.

Contents

1 Introduction

Parallel functional languages have a long history of being used for experimenting with novel parallel programming paradigms. Haskell, which we focus on in this paper, has several mature implementations. We regard here in-depth Glasgow parallel Haskell or short GpH (its Multicore SMP implementation, in particular), the *Par* Monad, and Eden, a distributed memory parallel Haskell. These languages represent orthogonal approaches. Some use a Monad, even if only for the internal representation. Some introduce additional language constructs. Section **??** gives a short overview over these languages.

A key novelty in this paper is to use Arrows to represent parallel computations. They seem a natural fit as they can be thought of as a more general function arrow ($\rightarrow$) and serve as general interface to computations while not being as restrictive as Monads (**?**). Section **??** gives a short introduction to Arrows.

We provide an Arrows-based type class and implementations for the three above mentioned parallel Haskells. Instead of introducing a new low-level parallel backend to imple-

ment our Arrows-based interface, we define a shallow-embedded DSL for Arrows. This DSL is defined as a common interface with varying implementations in the existing parallel Haskells. Thus, we not only define a parallel programming interface in a novel manner—we tame the zoo of parallel Haskells. We provide a common, very low-penalty programming interface that allows to switch the parallel implementations at will. The induced penalty was in the single-digit percent range, with means typically under 2% overhead in our measurements over the varying cores configuration (Section **??**). We have also conducted some initial experiments with Hdph and Frege. These experiments inducate that our parallel Arrows DSL can be relatively easily extended to other languages.

Contributions. We propose an Arrow-based formalism for parallelism based on a new Arrow combinator $parEvalN :: [arr\ a\ b] \to arr\ a\ [b]$. A parallel Arrow is still an Arrow, hence the resulting parallel Arrow can still be composed in the same way as a potential sequential version. In this paper we evaluate the expressive power of such a formalism in the context of parallel programming.

- We introduce a parallel evaluation formalism using Arrows. One big advantage of our specific approach is that we do not have to introduce any new types, facilitating composability (Section **??**).
- We show that PArrow programs can readily exploit multiple parallel language implementations. We demonstrate the use of GpH, a *Par* Monad, and Eden. We do not re-implement all the parallel internals, as we host this functionality in the *ArrowParallel* type class, which abstracts all parallel implementation logic. The implementations can easily be swapped, so we are not bound to any specific one.
  This has many practical advantages. For example, during development we can run the program in a simple GHC-compiled variant using GpH and afterwards deploy it on a cluster by converting it into an Eden program, by just replacing the *ArrowParallel* instance and compiling with Eden's GHC variant (Section **??**).
- We extend the PArrows formalism with *Future*s to enable direct communication of data between nodes in a distributed memory setting similar to Eden's Remote Data (*RD*, **?**). Direct communication is useful in a distributed memory setting because it allows for inter-node communication without blocking the master-node. (Section **??**)
- We demonstrate the expressiveness of PArrows by using them to define common algorithmic skeletons (Section **??**), and by using these skeletons to implement four benchmarks (Section **??**).
- We practically demonstrate that Arrow parallelism has a low performance overhead compared with existing approaches, e. g. the mean over all cores of relative mean overhead was less than 3.5% and less than 0.8% for all benchmarks with GpH and Eden, respectively. As for *Par* Monad, the mean of mean overheads was in our favour in all benchmarks (Section **??**).

PArrows are open source and are available from `https://github.com/s4ke/Parrows`.

2 Related Work

We chose to not extend our implementation to LVish and HdpH at the moment. The former does not differ from the original *Par* Monad with regard to how we would have used it

in this paper. The latter heavy relies on Template Haskell, which adds complexity to the implementation.

Parallel Haskells. The non-strict semantics of Haskell, and the fact that reduction encapsulates computations as closures, makes it relatively easy to define alternate parallelisations. A range of approaches have been explored, including data parallelism (**??**), GPU-based approaches (**??**), software transactional m[illegible] (**??**). The Haskell–GPU bridge Accelerate (**???**) is completely orthogonal to ou[illegible]pproach. [illegible]d survey of parallel Haskells can be found in **?**.

Our PArrow implementation u[illegible]ree task par[illegible]anguages as backends: the GpH (**??**) parallel Haskell dialect and its [illegible]ulticore vers[illegible]), the *Par* Monad (**??**), and Eden (**??**). These languages are under active develop[illegible] for example a combined shared and distributed memory implementation of Gp[illegible]s available (**??**). Research on Eden includes low-level implementation (**??**), skele[illegible] composition (**?**), communication (**?**), and generation of process networks (**?**). The de[illegible]itions of new Eden skeletons is a specific focus (**????????**).

Other task parallel Haskells related to E[illegible]en, GpH, and the *Par* Monad include the following. HdpH (**??**) is an extension of *Par* Monad to heterogeneous clusters. LVish (**?**) is a communication-centred extension of *Par* M[illegible]ad.

Algorithmic skeletons. Algorithmic skeletons were introduced by **?**. Early publications on this topic include (**?????**). **?** consolidated early reports on high-level programming approaches. Types of algorithmic skeletons include *map*-, *fold*-, and *scan*-based parallel programming patterns, special applications such as divide-and-conquer or topological skeletons.

The *farm* skeleton (**???**) is a statically task-balanced parallel *map*. When tasks' durations cannot be foreseen, a dynamic load balancing (*workpool*) brings a lot of improvement (**?????**). For special tasks *workpool* skeletons can be extended with dynamic task creation (**???**). Efficient load-balancing schemes for *workpool*s are subject of research (**??????**). The *fold* (or *reduce*) skeleton was implemented in various skeleton libraries (**????**), as also its inverse, *scan* (**??**). Google *map–reduce* (**??**) is more special than just a composition of the two skeletons (**??**).

The effort is ongoing, including topological skeletons (**?**), special-purpose skeletons for computer algebra (**????**), iteration skeletons (**?**). The idea of **?** is to use a parallel Haskell to orchestrate further software systems to run in parallel. **?** compare the composition of skeletons to stable process networks.

Arrows. Arrows were introduced by **?** as a less restrictive alternative to Monads, in essence they are a generalised function arrow $\rightarrow$. **?** presents a tutorial on Arrows. **???** develop theoretical background of Arrows. **?** introduced a new notation for Arrows. Arrows have applications in information flow research (**???**), invertible programming (**?**), and quantum computer simulation (**?**). But probably most prominent application of Arrows is Arrow-based functional reactive programming, AFRP (**???**). **?** formally define a more special kind of Arrows that capsule the computation more than regular Arrows do and thus enable optimisations. Their approach would allow parallel composition, as their special Arrows