View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by University of Szeged

Parallelisation of Haskell Programs by Refactoring

Norbert Luksa, Tamás Kozsik

Abstract: We propose a refactoring tool for the Haskell programming language, capable of introducing parallelism to the code with reduced effort from the programmer. Haskell has many ways to express concurrency and parallelism. Moreover, Eden, a dialect of Haskell supports a wide range of features for parallel and distributed computations. After comparing a number of possibilities we have found that the Eval Monad, the Par Monad and the Eden language provide similar parallel performance. Our tool is able to introduce parallelism by turning certain syntactic forms into the application of algorithmic skeletons, which are implemented with the Eval Monad, the Par Monad and the Eden language.

Keywords: Parallel Programming, Haskell, Eden, Par Monad, Eval Monad, Refactoring, Haskell-tools

Introduction

When it comes to programming, there are various factors to keep in mind. First of all, the program should accomplish the given task, but in the meantime it should be also efficient, furthermore the code should be clean and readable. Focusing on the speed, we have several possibilities for improvement. We often try to write efficient programs, which the compiler can further enhance with optimizations, but after a point there is no way to speed up a sequential program: this is when parallelism comes to mind.

The *Haskell* programming language has a wide range of possibilities to express parallelism. We have analyzed some of these methods, such as the *Eval Monad* with its parallel evaluation strategies, the *Par Monad*, and *Eden*, an extension of *Haskell*, a language with constructs for the definition and instantiation of parallel processes. Our goal was to find the best among these not only in terms of speed up, but also while looking at the ways these methods support parallelism – hence our final goal is to turn a sequential program to parallel with a refactoring tool, with as little effort from the programmers' side as possible.

Options for Parallelism

As part of our research we have examined various methods to describe parallelism. From these, we have compared three approaches in more detail through case studies of different complexity.

Evaluations Strategies

Pure parallelism in *Haskell* is achieved using only two primitives:

```
1 par  :: a -> b -> b
2 pseq :: a -> b -> b
```

While `par` indicates that it may be beneficial to lazily evaluate the first argument in parallel with the second, `pseq` forces this evaluation to actually take place. Although they are not too convenient to use, with an extra layer of abstraction they suddenly become a great and simple tool to express parallelism. This is achieved by introducing the simple idea of *Evaluation Strategies*, or *Strategies* for short [3]. A Strategy is a function with the following simplified type:

```
1 type Strategy a = a -> Eval a
2 data Eval a = Done a
3 runEval :: Eval a -> a
```

Then we can define some basic strategies, like the monadic counterparts of the functions above, and some more complex ones, such as `parList` – which evaluates the elements of a list in parallel –, and many others. There is a great thing with these Strategies when we think about refactoring: a simple construct lets us turn a sequential program into parallel without changing anything else in the code. The example below shows how to create a function, which would map a function over a list in parallel:

```
1 using :: a -> Strategy a -> a
2 x `using` strat = runEval (strat x)
3 parMap f xs = map f xs `using` parList rseq
```

The Par Monad

Another useful tool is the Par Monad [4], which is built around the concept of *forking*: the computation passed as the argument to function `fork` is computed in parallel. We can see this as a *parent-child* evaluation, where the passed argument is computed in the child process, while the current computation is executed in the parent process.

```
1 fork :: Par () -> Par ()
```

Adding once again an extra layer over this simple basic construct, we can have another great tool. The communication among the processes is accomplished through `IVar`s (I-structures), which are write-once mutable reference cells with the operations `put` and `get`. Function `put` assigns a value to an `IVar`, while `get` returns the value after the assignment takes place (and blocks until this happens).

```
1 new :: Par (IVar a)
2 get :: IVar a -> Par a
3 put :: NFData a => IVar a -> a -> Par ()
```

By defining a common pattern that creates a child and then collects the result, `spawn`, we can easily give the definition of `parMap` again, now using the Par Monad.

Eden

Eden is a parallel functional language, extending Haskell with constructs for creating and computing processes in parallel [1].

```
1 process :: (Trans a, Trans b) => (a -> b) -> Process a b
2 (#)     :: (Trans a, Trans b) => Process a b -> a -> b
```

Here `process` creates the *process abstraction*, while `(#)` instantiates it. The evaluation of an expression like `(process func) # arg` leads to the evaluation of the argument `arg` on a new thread in the parent process and will be sent to the child process which returns the result of `func arg` in a demand driven way. The type context `Trans a` means that `a` has to be *transmissible*. The two constructs described above are usually used in a combination resulting in parallel function application:

```
($#) :: (Trans a, Trans b) => (a -> b) -> a -> b
f $# x = process f # x
```

So once again we have the basic construct – but we can add an extra layer of abstraction to have a convenient way of writing a parallel program.

Comparison

We have tested the performance of the above mentioned approaches on simple and more complex problems [2, 5]. The achieved speed-up lived up to our expectations with all three approaches showing good results. We also had to consider other factors as well, most importantly how complex it is with the three approaches to introduce parallelism as code transformations to a sequential program.

After several case studies, we have found the three different methods equally convenient, so we decided to support all the three. However, since *Eden* and its predefined algorithmic skeletons provided a very nice abstraction layer, we have implemented these skeletons using the other two approaches in order to provide a common ground for the parallelisation code transformations.

General skeletons

To generalize over the three considered parallelization methods, we have introduced general algorithmic skeletons. We have decided to define these skeletons in a language-independent manner. Our restriction only lies in the necessary methods, such as *map* or *parMap*. Given these functions, we can specify different algorithms in a general way on any traversable structures.

Function *parMap* can be considered the simplest skeleton, which takes a list and creates a process executed in parallel for each of its elements. Of course, this naive version cannot be used in a general problem where the length of a list is probably larger, large enough that the communication among the processes would result in severe overhead. An improvement in our case is the *farm* skeleton which distributes its input to a number of worker processes, just like in the *Eden* version. For this, we need two new functions: one which distributes the elements, and another one which combines the results. Traversable data structures will be denoted with curly braces, such as `{Int}`.

$$\begin{aligned} Distribute &\triangleright \{a\} \to \{\{a\}\} \\ Combine &\triangleright \{\{a\}\} \to \{a\} \end{aligned}$$

$$\frac{Combine \circ Distribute \equiv Id, \quad f \triangleright a \to b}{Farm(f) \equiv Combine \circ ParMap(Map(f)) \circ Distribute}$$

Observe that we can allow the inside and outside structures in the domain of *Combine* and in the co-domain of *Distribute* to be different: we can split a tree into a list of trees, for instance.

There are quite a few common parallel patterns which we can express with the general skeletons. We have given definitions for reduction, for map-reduce, and for divide-and-conquer schemes.

Transformation to expressions

For each of our general skeletons we have defined their transformations to abstract expressions. At this stage we have still stayed with the general-purpose solution, but we have also kept in mind that we want to introduce code transformations in the *Haskell Tools* refactoring environment [6]. Therefore we use expressions as represented in this tool. In order to have a better understanding of these expressions, consider the following example.

$$\begin{aligned} 3+2 &\triangleright Expr \\ 3+2 &\triangleright Expr \to Expr \to Expr \to Expr \end{aligned}$$

Here $3 + 2$ can be considered a simple expression, or we can look at it as a combination of three expressions creating a new one. However, if we introduce the pattern of *Operator* on the $(+)$ sign, we can see the type signature of an infix application. If we introduce its pattern, we can give the type of the addition as follows.

$$\begin{aligned} InfixApp &\equiv Expr \to Operator \to Expr \to Expr \\ 3+2 &\triangleright InfixApp \end{aligned}$$

We have defined the syntactic occurrence of algorithmic skeletons using the Haskell Tools' representation of Abstract Syntax Trees.

Refactoring

In order to provide a convenient way to refactor sequential Haskell programs into parallel, we have extended *Haskell Tools*, a refactoring environment for *Haskell* programs [6] with support for parallelization transformations.

Transforming a sequential code to parallel is just the first step. Ensuring that the parallel code runs faster than the sequential one is also difficult. The costs of process creation and communication may cancel out the performance gains of parallel evaluation. A tool should help the developer make decisions on which parallel skeleton, if any, to use in a given situation to achieve significant efficiency improvement. Cost-directed refactoring [7, 8, 9] uses cost models to facilitate such decisions – our future work will target this approach.

Acknowledgement

Norbet Luksa was supported by ÚNKP-17-1 New National Excellence Program of The Ministry of Human Capacities (Hungary). Tamás Kozsik was supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013)

References

[1] Loogen, R. (2011). Eden – Parallel Functional Programming with Haskell. In: *Proc. 4th Central European Functional Programming School*, pp. 142–206. Springer

[2] Marlow, S. (2011). Parallel and Concurrent Programming in Haskell. In: *Proc. 4th Central European Functional Programming School*, pp. 339–401. Springer

[3] Marlow, S., Maier, P., Loidl, H., Aswad, M., Trinder, P. (2010). Seq no more: better strategies for parallel Haskell. In: *Proc. 3rd ACM Symposium on Haskell*, pp. 91–102. ACM

[4] Marlow, S., Newton, R., Jones, S.P. (2011). A monad for deterministic parallelism. *SIGPLAN Not.* **46**(12):71–82.

[5] Loidl, H., Totoo, P. (2014). Parallel Haskell implementations of the N-body problem. *Concurrency and Computation: Practice and Experience* **26**:987–1019.

[6] Németh, B. *Haskell-tools*. Retrieved from https://github.com/haskell-tools/haskell-tools

[7] Brown, C., Danelutto, M., Elliott, A., Hammond, K., Kilpatrick, P. (2013). Cost-Directed Refactoring for Parallel Erlang Programs. *International Journal of Parallel Programming* **42**:564–582.

[8] Berthold, J., Hammond, K., Loogen, R. (2003). Automatic Skeletons in Template Haskell. *Parallel Processing Letters* **13**:413–424.

[9] Loogen, R., Ortega, Y., Peña, R., Priebe, S., Rubio, F. (2003). Parallelism abstractions in Eden. In: *Patterns and Skeletons for Parallel and Distributed Computing*, pp. 95–128. Springer