Exploring Parallel Programming in Haskell

Rashad Gover

May 11, 2021

Contents

Abstract

In this report, we will explore parallel computation in the pure functional language ***Haskell***.

1 Brief Introduction to Haskell

Haskell is a declarative, functional programming language invented in 1971 by a group of programming language researchers for the purpose of exploring the design space of functional languages. Unlike imperative languages which are based on turing machines and embrace state, Haskell and other pure functional languages are based on the lambda calculus, invented by Turing's professor Alonzo Church, where state is dropped in favor of mathematical functions. At a high-level, a Haskell program is just a function that is composed of many smaller functions. On top of being a functional programming language, Haskell is also a strongly, statically typed language which eliminates an entire class of errors (type errors) at compile-time.

Haskell has all the basic features that you would expect from any programming language like variables, function declaration, and a module system for decoupling code. Haskell also has more interesting features like type inferenece, user defined types, pattern matching, type classes, lazy evaluation by default, and monads for modelling side-effects without compromising purity. These traits, besides laziness, make Haskell a prime candidate for implementing parallel algorithms. The declarative nature of Haskell allows the user to focus on the problem at hand instead of the low-level details such as state, the order in which statements are declared, and memory management. Much of the burden that is usually placed on the shoulders of the programmer is pushed to the advanced compiler, which makes Haskell more ”ergonomic” in my opinion. To show this, let's implement a program in C++ and Haskell that takes in an array or list, repspectively, and increments each value, doubles it, and then prints the values. The implementation in C++ would look something like this:

Listing 1: C++ example

```cpp
#include <iostream>
int main() {
  int arrLength = 5;
  int arr [] = {1, 2, 3, 4, 5};
  for (int i = 0; i < arrLength; i++) {
    arr[i] += 1;
    arr[i] *= 2;
    std::cout << arr[i] << " ";
  }
  return 0;
}
```

The Haskell implementation would look like so:

Listing 2: Haskell example

```haskell
module Main where
list = [1, 2, 3, 4, 5]
incThenDouble x = (x + 1) * 2
```

```haskell
answer = map incThenDouble list
main = print answer
```

As we can see, the Haskell implementation is much more concise than the C++ implementation. You may also notice that compared to the Haskell version, the C++ version looks much like a ”recipe” where each step of transforming the array in to another must be described in detail: how long the array is, how to loop through the array, the types of the variables, etc. The Haskell version on the other hand is very declarative, and as a result looks a lot cleaner and readable. Obviously, this is a contrived example, but in my experience it seems to be the case more often than not that Haskell code looks more approachable compared to an imperative language.

1.1 Fibonacci Sequence

This section will go over the code in Haskell for the Fibonacci Sequence

Listing 3: Haskell Fibonacci

```haskell
module Main where
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

1.2 Matrix Multiplication

This section will go over the code in Haskell for Matrix Multiplication

Listing 4: Haskell Fibonacci

```haskell
module Main where

import Data.List

 mmult :: Num a => [[a]] -> [[a]] -> [[a]]
 mmult a b = [ [ sum $ zipWith (*) ar bc | bc <- (transpose b) ] | ar <- a ]
```

1.3 Benchmarking

This section goes over Threadscope and benchmarks the two examples above. Fibonacci and Matrix Mutiplication.

2 Parallel Evaluation Strategies

This section will breifly explain Strategies and the Eval monad, and the functions rpar and rseq.

2.1 Results

This section will have the results for the Fibonacci Sequence and Matrix Multiplication using Strategies.

3 Dataflow Parallelism with monad-par

This section will introduce the Par Monad and the concept of dataflow parallelism.

3.1 Results

This section will have the results for the Fibonacci Sequence and Matrix Multiplication usingthe Par Monad.

4 Accelerate Library

This section goes over the Accerlerate library and the idea of embedded DSLs withing Haskell, for which it is known for. In this section, HW2 particle simulation is used

4.1 Particle Simulation

This section describes the Particle Simulation algorithm in some detail and HW2.

4.2 Multicore Programming with Accelerate

This section goes over multicore programming in Accelerate.

4.2.1 Results & Comparison with OpenMP

In this section we compare the results with the OpenMP variant of the HW.

4.3 GPU Programming with Accelerate

This section goes over GPU programming in Accelerate. This will be pretty much the same as the multicore programming in the above section because we can write programs for both platforms with a single syntax! The syntax used to describe the parallel program is decoupled from the platform it runs on which is great.

4.3.1 Results & Comparison with CUDA

This section will go over the benchmark results and compare it with the CUDA implementation of HW2.

5 Conclusion

This section will rap up the results of the benchmarks and come to conclusion. Was climbing up the abstraction ladder worth the tradeoff in performance? Was the drop in performance of the Haskell implementations worth the better ergonomics? This could be argued, especially if the user isn't familiar with functional programming. Was the performance of the Haskell programs not that much of a difference from the C++ implementations? What do overhead costs look like? How does scaling compare between the two platforms? Weak and strong scaling.