

Exploring Parallel Programming in Haskell

Rashad Gover

May 13, 2021

Contents

1	Brief Introduction to Haskell	3
1.1	Fibonacci Sequence	4
1.2	Matrix Multiplication	4
1.3	Benchmarking	5
2	Parallel Haskell	5
2.1	Dataflow Parallelism with monad-par	5
2.1.1	Fibonacci Implementation	6
2.1.2	Results	6
2.2	Regular Parallel Arrays	6
2.2.1	Matrix Multiplication Implementation	6
2.2.2	Results	7
2.3	Particle Simulation with Accelerate Library	7
2.3.1	Particle Simulation	7
2.3.2	Multicore Results	7
2.3.3	GPU Results	7
3	Honorable Mentions	8
4	Conclusion	8
5	Prior Work & Resources	8

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 inference, 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, respectively, and increments each value, doubles it, and then prints the values. The implementation in C++ would look something like this:

Listing 1: C++ example

```
1 #include <iostream>
2 int main() {
3     int arrLength = 5;
4     int arr [] = {1, 2, 3, 4, 5};
5     for (int i = 0; i < arrLength; i++) {
6         arr[i] += 1;
7         arr[i] *= 2;
8         std::cout << arr[i] << " ";
9     }
10    return 0;
11 }
```

The Haskell implementation would look like so:

Listing 2: Haskell example

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

```
4 answer = map incThenDouble list
5 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

The Fibonacci sequence is a series of numbers where a number is the addition of the last two numbers, starting with 0, and 1. The serial implementation in Haskell is very straightforward. The following isn't the fastest implementation, but is probably the most idiomatic:

Listing 3: Haskell Fibonacci

```
1 module SerialFib where
2
3 nfib :: Int -> Int
4 nfib 0 = 0
5 nfib 1 = 1
6 nfib n = nfib (n-1) + nfib (n-2)
```

We pattern match on the value of the integer: if the value is 0 return 0, and if it is 1 return 1. Otherwise, return the sum of fib of the previous fibonacci number and the fibonacci before the previous one.

The use of recursion has some performance implications. The memory performance of recursive functions in Haskell differs depending on whether or not the recursive function is *tail recursive*. A recursive function is tail recursive if the final result of the recursive call is the final result of the function itself. If the result of the recursive call must be further processed (say, by adding 1 to it, or consing another element onto the beginning of it), it is not tail recursive. So in the case above, the function `nfib` is not tail recursive.

1.2 Matrix Multiplication

To represent matrix multiplication in Haskell, we use the `Vector` data type over the more idiomatic `List` data type. This is to enhance the performance of the serial program since `List` in Haskell is a lazy data structure, while a `Vector` value represents an *unboxed array* under the hood and thus has an $O(1)$ length operation and data access.

Listing 4: Haskell Fibonacci

```
1 module SerialMatMul where
2
3 import Data.Vector (Vector(..), (!))
```

```

4 import qualified Data.Vector as Vector
5
6 matMul
7   :: (Eq a, Num a)
8   => Vector (Vector a)
9   -> Vector (Vector a)
10  -> Either String (Vector (Vector a))
11 matMul a b
12   | Vector.length a == 0 || Vector.length b == 0 = Left "Empty matrices can't be used"
13   | not (isConsistentMatrix a) || not (isConsistentMatrix b) = Left "The dimensions"
14   | numOfCol a /= numOfRow b = Left "These matrices are incompatible for multiplication"
15   | otherwise = Right $ mul a b
16   where
17     numOfCol m = Vector.length $ Vector.head m
18     numOfRow m = Vector.length m

```

Ideally, to leverage the advanced type system of Haskell we would encode the length of the vectors in their types. The compiler would then be able to automatically handle any cases in which the dimensions of the matrices represented as vectors don't match up correctly. This is known as *type-level programming* and it allows the programmer to encode information in types to provide more information for the compiler and further constrain the number of unwanted outcomes. I opted for the above implementation due to ease of implementation and better performance, but needed to add guards to check the dimensions of the matrices at the value-level.

1.3 Benchmarking

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

2 Parallel Haskell

There are various ways of expressing parallelism in Haskell ranging from monad-based libraries like `monad-par` to embedded DSLs (Domain Specific Languages) like `accelerate`. These libraries can be vastly different from one another, so the programmer must know how to choose the right tool for the job. Certain libraries will only really make sense for certain types of problems. To test these various methods of expressing parallelism, I've created a series of microbenchmarks to test them and compare them with serial versions of the same program.

2.1 Dataflow Parallelism with `monad-par`

Dataflow parallelism is best used for implementing parallel algorithms where the problem can be broken down into smaller pieces and solved recursively. Problems that would fit in this category are generating the *n*th Fibonacci number, mergesort, or traversing a binary tree. The best library I found for expressing dataflow parallelism is `monad-par`. Monads are a *class of types* that are commonly used in Haskell as a way to model functions that produce effects. Monads can express many different types of effects, such as non-determinism, state, IO with the outside world, and also parallelism. Without monads,

these sort of effectful functions couldn't exist in a pure functional language like Haskell. The `monad-par` library defines a monad for the user through which they can express parallel computation.

2.1.1 Fibonacci Implementation

Since `monad-par` is great for problems that can be recursively broken down and solved, it is a perfect fit for parallelizing the `nfib` function defined earlier:

Listing 5: Fibonacci using `monad-par` library

```
1 module MonadParFib where
2
3 import Control.Monad.Par
4
5 nfib :: Int -> Par Int
6 nfib n = do
7     firstComputation <- spawn (nfib (n - 1))
8     secondValue <- nfib (n - 2)
9     firstValue <- get firstComputation
10    return (firstValue + secondValue)
```

The `spawn` function forks a new thread where the calculation of the previous Fibonacci number happens. Then, the result of a second call to `nfib` is produced. While the second value is being calculated, the first computation is still running or maybe already done. Once the second value has been produced, the value of the first computation must be extracted to use it. Finally, the two results are combined to produce the final answer. It helps to think of it as branching tree, where each call to `nfib` produces two more branches that run in parallel.

2.1.2 Results

2.2 Regular Parallel Arrays

In contrast to dataflow parallelism which is great for recursive problems, Haskell has an array-based library called `repa` that is great for any problems that can be modelled as an array of data. `repa` gives you the ability to represent polymorphic, parallel arrays in Haskell, which is a big step up from the more commonly used, but less performant `List` or `Vector` data types. Any functions that use the data structures and special operators provided by the `repa` library are automatically parallelized by simply passing in a flag to GHC (Glasgow Haskell Compiler). This makes it easy to parallelize serial code that use sequential data structures inefficiently.

2.2.1 Matrix Multiplication Implementation

The `repa` library is an obvious fit for parallelizing matrix multiplication. Instead of using a nested `Vector` data type to represent a matrix, we can use the array shapes provided by the 'repa' library:

Listing 6: Matrix multiplication using `repa` library

```
1 module RepaMatMul where
2
```

```

3 import Data.Array.Repa (Array(..), U(..), Z(..), DIM2(..), (:.)(..))
4 import Data.Array.Repa as Repa
5 import Data.Vector.Unboxed.Base (Unbox(..))
6
7 matMul
8   :: (Monad m, Num a, Unbox a)
9   => Array U DIM2 a
10  -> Array U DIM2 a
11  -> m (Array U DIM2 a)
12 matMul a b = Repa.sumP (Repa.zipWith (*) aNew bNew)
13   where
14     bTranspose = Repa.transpose b
15     (Z :. aCols :. aRows) = Repa.extent a
16     (Z :. bCols :. bRows) = Repa.extent b
17     aNew = Repa.extend (Z :. All :. bCols :. All) a
18     bNew = Repa.extend (Z :. aRows :. All :. All) bTranspose

```

The nice thing about this implementation is that not only is it more performant than the serial implementation introduced earlier, but it also more *type safe*, meaning that the compiler can better catch our mistakes because it knows more about the program at compile-time. If you look at the type signature of the `matMul` function, you can see that it provides the shape of the array and the exact dimensions of the array. In our case, we are using an array of 2 dimensions, so the `Array` type is given the type `DIM2` as an argument to represent this. These more detailed types also allow the Haskell runtime to make better optimizations to the code.

2.2.2 Results

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

2.3 Particle Simulation with Accelerate Library

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

2.3.1 Particle Simulation

2.3.2 Multicore Results

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

2.3.3 GPU Results

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.

3 Honorable Mentions

4 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.

5 Prior Work & Resources