Exploring Parallel Programming in Haskell

Rashad Gover

May 13, 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 1987 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 good 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. The idea is that this same conciseness will carry over to parallel programs.

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

```haskell
module SerialFib where

nfib :: Int -> Int
nfib 0 = 0
nfib 1 = 1
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 number 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, 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 Haskel 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 MatMul

```haskell
module SerialMatMul where

import Data.Vector (Vector(..), (!))
import qualified Data.Vector as Vector

matMul
  :: (Eq a, Num a)
  => Vector (Vector a)
  -> Vector (Vector a)
  -> Either String (Vector (Vector a))
matMul a b
  | Vector.length a == 0 || Vector.length b == 0 = Left "Empty matrices can't be us
  | not (isConsistentMatrix a) || not (isConsistentMatrix b) = Left "The dimensions
  | numOfCol a /= numOfRow b = Left "These matrices are incompatible for multiplica
  | otherwise = Right $ mul a b
  where
    numOfCol m = Vector.length $ Vector.head m
    numOfRow m = Vector.length m
```

Ideally, to leverage the advanced type sytem 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

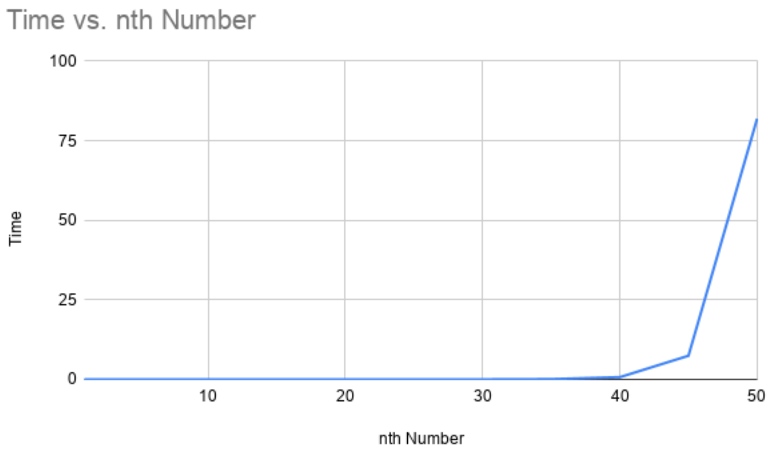
All benchmarks were recorded on my personal laptop. The machine used for all tests has 4 cores and 16GB RAM. I'm using some simple timer functions to get the times for my data, so they may not be the most accurate results:

Listing 5: Clock functions

```haskell
currentTime = getCurrentTime

timeElapsedSince t0 = do
  t1 <- getCurrentTime
  printf "time: %.2fs\n" (realToFrac (diffUTCTime t1 t0) :: Double)
```

The results for the serial version of Fibonacci were interesting:



The function was able to able to handle calculating the Fibonacci numbers in a reasonable amount of time up until I tried to ask it for the 40th Fibonacci number and beyond. After asking for the 45th number, the time it took to come up with an answer increased exponentially. Asking for the 50th Fibonacci number took 81.94 seconds!

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 will 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 nth 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. With out 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 6: Fibonnaci using monad-par library

```haskell
module MonadParFib where

import Control.Monad.Par

nfib :: Int -> Par Int
nfib n = do
  firstComputation <- spawn (nfib (n - 1))
  secondValue <- nfib (n - 2)
  firstValue <- get firstComputation
  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.

It also important to note that the runtime of the 'Par' monad is based on *sparks* which are different than threads in the context of Haskell, where "threads" usually refer to green threads.

2.1.2 Results

Unfortunately, I was unable to obtain any useful results from benchmarking this function. My computer crashed even when asking for the 1st Fibonacci number, so it did really bad compared to the serial version, which is not what I expected. Trying to run the program on any amount of cores, from 1 - 4, produced no results. Each time the program ended with `Killed` as the output. As far as I know, this means that the runtime ran out of memory or got hung up at a certain point in the program. I honestly don't know what went wrong here. My theory is that my machine is too weak, or the overhead of using `monad-par` is too much. From my research, similar examples with branching logic seemed to work fine, so I'm not sure if it just has to do with my machine. Since the Haskell runtime is really complex, it's more likely that I missed an opportunity to make some part of the program strict instead of lazy, which Haskell is by default. I've been burned in the past by Haskell's laziness, so this wouldn't be the first time.

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. `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 ineffeciently.

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 7: Matrix multiplication using repa library

```haskell
module RepaMatMul where

import Data.Array.Repa (Array(..), U(..), Z(..), DIM2(..), (:.)(..))
import Data.Array.Repa as Repa
import Data.Vector.Unboxed.Base (Unbox(..))

matMul
  :: (Monad m, Num a, Unbox a)
  => Array U DIM2 a
  -> Array U DIM2 a
  -> m (Array U DIM2 a)
matMul a b = Repa.sumP (Repa.zipWith (*) aNew bNew)
    where
      bTranspose = Repa.transpose b
      (Z :. aCols :. aRows) = Repa.extent a
      (Z :. bCols :. bRows) = Repa.extent b
      aNew = Repa.extend (Z :. All :. bCols :. All) a
      bNew = Repa.extend (Z :. aRows :. All :. All) bTranspose
```

The nice thing about this implementation is that not only is it more performant that 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

I also failed to obtain any useful results from benchmarking this function either, which is a shame because I spent a lot of time implementing this one. My theory is that I was unable to effeciently convert from the `List` data type to the `repa` data types that are required for the parallelism to work. After discussion with community memebers, it seems that the function should work correctly, but the IO that I perform to setup the function for testing maybe the bottleneck.

2.3 Particle Simulation with Accelerate Library

Implementing a particle simulator in Haskell using the `accelerate` library was the main focus of the paper, but like my `monad-par` and `repa` implementations, I was unable to obtain any meaningful results. I reached out to the official maintainer of the library to see what I could do, but received no response. The community reccomended that I use another similar library called `massiv` to see what happens. I was too deep in and decided to go forward with what I had, but ultimately came to no conclusion on whether or not `accelerate` is effective for particle simulation. I was inspired by the examples provided by the `accelerate` library and everything type-checked and compiled, but was I unable to benchmark the simulation function and get useful results.

3 Honorable Mentions

There were some other paths that I went down on my journey to proving Haskell useful in a parallel computing context. Besides the `monad-par` library, there is a similar library called `strategies` that cought my eye early on, but I dropped it due to its' similarity to `monad-par`. Now I wonder if going with `strategies` would've produced a better outcome, but I can't be sure.

Another worthy mention are a group of functions I implemented for microbenchmarking called `checkAll` (serial) and `checkAllPar` (parallel) which check to see if every value in a binary tree satisfy a given predicate. I used the `monad-par` library to implement this due to its' branching, recursive nature and ran into the same issues I ran into with my implementation of the Fibonnaci sequence solver. Since it is nearly identical to my Fibonacci example, I left it out to pursue the more interesting aspects of the project, but feel free to check out the repository code if you're interested in seeing how it works. Here's what the code for those functions looks like:

Listing 8: Serial Tree check

```haskell
module Tree.SerialTree where

data Tree a = Leaf a
            | Node (Tree a) (Tree a)
  deriving (Eq, Show)

-- Check that all values in the tree satisfy a given predicate
checkAll :: (a -> Bool) -> Tree a -> Bool
checkAll pred (Leaf value) = pred value
checkAll pred (Node leftTree rightTree) = (checkAll pred leftTree)  (checkAll pred
```

Listing 9: Tree check using monad-par

```haskell
module Tree.MonadParTree where

import Tree.SerialTree (Tree(..))
import Control.Monad.Par

-- Test whether all elements match the given predicate (in parallel)
checkAllPar :: (a -> Bool) -> Tree a -> Par Bool
checkAllPar pred (Leaf value) = return $ pred value
checkAllPar pred (Node leftTree rightTree) = do
  leftTreeComputation <- spawn $ checkAllPar pred leftTree
  rightTreeResult  <- checkAllPar pred rightTree
  leftTreeResult  <- get leftTreeComputation
  return $ leftTreeResult  rightTreeResult
```

4 Conclusion

All in all, I can truly say that my initial hypothesis of Haskell being a more "ergonomic" language than a traditional imperative language for expressing parallel algorithms was wrong. Yes, the code may be more concise and prettier, but the setup and tooling around parallel programming seems to be lacking when compared to other languages. I found little resources and knowledge on benchmarking and reasoning about parallelism in Haskell. As a Haskell developer of a couple years now, even reasoning about serial code can be difficult, but I didn't expect parallelism in Haskell to be this much harder.

Another pain point I experienced was laziness. Laziness makes predicting what the runtime is going to do a lot more difficult and it really hurt me. In the back of my head, I still wonder if some of my benchmarks weren't working simply because of a failure to reduce a thunk or use a strict data type. Laziness does provide by benefits, like forcing the language to be pure and having the ability to produce streams of data easily, but for parallel programming lazy evaluation is definitely a pain.

Last but not least, I failed on my part to find a strong machine to test my programs out on. I was unable to setup the Haskell environment on CORI, so my laptop was the best option knowing what I knew at the time I started this project. If I could do it over again, I would use a machine on Digital Ocean or AWS to rule out any sort of issues that may have been caused by using a machine that is too weak.

The best things about the project were that I got to get more familiar with one of my favorite programming languages, and got to see parallel programming from another perspective that it isn't seen from often. So to answer to the question I initially had at the start of this journey, I would say that Haskell is most definitely not the best tool for parallel programming. I wasn't expecting it to be, but at the same time, I didn't expect the experience to be as bad as it was. At the end of the day, I must also consider my lack of skill and the fact that I wasn't able to devote 100% of my focus to this project as I would've liked. I hope one day I can revisit this project and give it the conclusion it deserves.

5 Prior Work & Resources

1. GitHub repo for the entire project: https://github.com/rashadg1030/cs267-final

2. Reference for GHC (Glasgow Haskell Compiler): https://www.haskell.org/ghc/

3. Book by Simon Marlow on Parallel and Concurrent Programming in Haskell: https://www.oreilly.com/library/view/parallel-and-concurrent/9781449335939/

4. Documentation for `monad-par` library: http://hackage.haskell.org/package/monad-par

5. A blog post on using the `monad-par` library: http://tomasp.net/blog/speculative-par-monad.aspx/

6. Documentation for `repa` library: https://hackage.haskell.org/package/repa-3.4.1.4/docs/Data-Array-Repa.html

7. A tutorial for using `repa`: https://wiki.haskell.org/Numeric_Haskell:_A_Repa_Tutorial

8. Documentation for `accelerate` library: https://hackage.haskell.org/package/accelerate-1.3.0.0/docs/Data-Array-Accelerate.html

9. A compilation of examples of `accelerate` in use: https://github.com/AccelerateHS/accelerate-examples

10. An academic paper on parallel evaluation strategies in Haskell: http://simonmar.github.io/bib/papers/strategies.pdf

11. Great resource for learning LaTeX: https://www.overleaf.com/