Exploration of Parallelism in Haskell

Miles Beverly, Alec Jackson
CS 4700
April 27, 2020

Summary

For our project on Parallelism in Haskell, we learned Haskell and have implemented two primary algorithms in our attempt to explore the nature of parallel computation in Haskell. We also researched lambda calculus, which is the underlying logical paradigm of all functional programming languages, and therefore of Haskell. We implemented each of the algorithms in both C++ and Haskell and compared the results. After examining data from these experiments, we found that while Haskell supports easier and more simplified implementation of parallelism, the overhead for ensuring thread-safe computation is too high, making it so there is no increase in performance by executing tasks in the algorithms we implemented in parallel.

Motivation

The purpose of this project was to learn more about functional programming and parallel programming. We were also interested in learning more about Haskell, which is probably the most well-known example of a language which is both purely functional and strongly typed.

Research

We focused our research on learning how to write Haskell code, how functional programming languages like Haskell work through lambda calculus in implementing parallel tasks, and the C++ Message Passing Interface (MPI). As neither of us had any experience coding in Haskell before beginning this project, our intent in researching these topics was to better our understanding of Haskell so we could proficiently implement our chosen algorithms and learn how parallelism works in this language. We also wanted to understand lambda calculus better to learn how parallel tasks might be computed in a functional language such as Haskell, and MPI in order to implement our code in another parallel language and compare the results.

Haskell and Lambda Calculus

While researching Haskell, we found it extremely beneficial to understand it better by learning more about lambda calculus. Lambda calculus was discovered by Alonzo Church in the 1930s, and is Turing complete, which is to say it can be used to simulate any Turing machine (https://brilliant.org/wiki/lambda-calculus/).

Haskell is a purely functional language and functions do not have side effects, which is the reason for its inherent parallelism (along with some other benefits). However, side

effects are necessary for some basic programming tasks such as input/output, reading from files, etc. To compensate for this, Haskell logically separates functions which are pure and functions which perform tasks such as input/output, allowing computations to be completely functionally pure, after which the non-pure functions can use the data returned by the pure functions.

In Haskell, like most purely functional languages, data is immutable. Once data has been defined, you cannot redefine it, although you can create a new data instance by the same name. This might seem to have the same effect as redefining data, but it doesn't quite, which can lead to some bugs when attempting to transition from imperative programming.

One interesting thing we learned about parallelism in Haskell is that you simply need to identify which expressions can be evaluated in parallel and the compiler will make a determination as to whether the cost of evaluating an expression in parallel is worth the parallel overhead. You can identify code which can execute in parallel in Haskell by importing Control.Parallel and marking the code you wish to run in parallel with the "par" or "pseq" functions. One thing we are discovering is that while it has generally seemed easier to implement parallel algorithms in Haskell, it doesn't mean that the parallel performance is actually better, due to the higher overhead Haskell requires. In fact, the documentation for the Haskell parallel programming package advises against attempting to run trivial operations in parallel (https://hackage.haskell.org/package/parallel-3.2.2.0/docs/Control-Parallel.html).

Another feature of Haskell we explored is its use of lazy evaluation, which means that expressions are not evaluated until they are needed. This allows programs to avoid wasting resources computing values which will never be needed, and also allows for infinite lists such as [1..], as each element of the list is only evaluated when it is needed.

The main resource we have been using to learn Haskell is "*Learn You a Haskell*" (http://learnyouahaskell.com/chapters). The website has a good reputation on Haskell forums on the web, and was very helpful in learning this new language.

C++ MPI

To compare parallelism in Haskell with a technology we are already familiar with, we decided to recreate our parallel Haskell algorithms in C++ using the Message Passing Interface (MPI). This message-passing standard works by the user specifying the number of threads to initialize, and each thread runs its portion of the code individually, while passing information through standardized data packets. With MPI, parallelism is much more manual than with Haskell, giving us a little more flexibility as to how the code is executed in parallel and with how many threads. We can specify exactly which threads execute which portions of code, whereas with Haskell, most of that is done implicitly and automatically.

Algorithms

The algorithms we intend to compare are a parallel global sum function and a parallel sorting function. The global sum function calculates the sum of an arbitrary amount of numbers and is implemented in a similar way in both Haskell and C++. For the sorting algorithm, we will compare a parallel quicksort algorithm implemented in Haskell to a parallel C++ quicksort. We intend to compare these algorithms using their runtimes. We will also compare them on conciseness (using lines of code) and readability (by giving them random names and seeing which ones some of our fellow computer science students find easier to understand).

Haskell Implementations
Here is the Haskell global sum function:

```
import Control.Parallel (par, pseq)

parGlobalSum arr
     | length arr == 0 = 0
     | length arr == 1 = arr !! 0
     | otherwise = upper `par` (lower `pseq` (lower +
upper))
     where
               lower = parGlobalSum [arr !! i | i <-
     [0..len `div` 2]]
               upper = parGlobalSum [arr !! i | i <- [len
     `div` 2 + 1..len]]
               len = length arr - 1
```

Here is the code for our Haskell quicksort algorithm:

```
module Main where

import Prelude
import Control.Parallel

quickSort [] = []
quickSort (x:xs) = small `par` (big `par`
    (small ++ [x] ++ big))
    where
        small = quickSort [p | p <-  xs, p <= x]
        big = quickSort [p | p <- xs, p > x]
```

C++ MPI Implementations

In C++ MPI, the code itself runs many lines so instead of providing the code here, we will explain how the algorithms work. All of our source files are included with the report as well.

The algorithms we chose to implement in C++ are a tree-structured global sum, which is implemented in much the same way as with Haskell, and a quicksort. The tree-structured global sum algorithm designates each thread to act as a “node” in the tree structure, similar to the structure of a binary search tree. Each thread, or node, generates a random value and sends it to its parent node. The “leaf” nodes begin this process, and then each parent node will send the sum of all of the values belonging to its sub-tree to its parent node. The root node then calculates the total sum and reports it.

The parallel quicksort algorithm is simply a quicksort algorithm in C++ that uses the MPI to send messages between the threads. First, the array is divided into parts by the master thread. Those parts are then sent to different processes. When each process gets its part, it sorts its part sequentially with a normal (serial) quicksort algorithm. After this, all the parts are sent back to the thread from which it came. The master thread eventually receives all the array partitions and compiles the fully-sorted array. To determine the array partitions, each process divides its portion of the array with the serial quicksort algorithm according to numbers which are less than or greater than the found pivot. The smaller half of the array is sent to the next available process. It keeps dividing the array until there are no more available processes, and then does a serial quicksort with the portion of the array that remains.

Results

As we began exploring Haskell and how parallel computation may occur in Haskell implementations, we expected to find that parallelism would be simple to write and simple to read. We also expected to find that in adding more threads, the programs would execute more quickly. While we did find that implementing parallelism in Haskell is fairly simple and easy, we also found that executing our test algorithms with multiple threads did not cause a significant increase in how quickly they completed.

Simplicity
When we initially wrote the code for our algorithms in both Haskell and C++, we found that it was much easier to implement parallelism in Haskell. Because data is immutable in Haskell, it is impossible to encounter race conditions in multi-threaded tasks. Parallelism is also implicit. The threads generate themselves and tasks are divided between the tasks automatically within the user-defined block of parallel executable code.

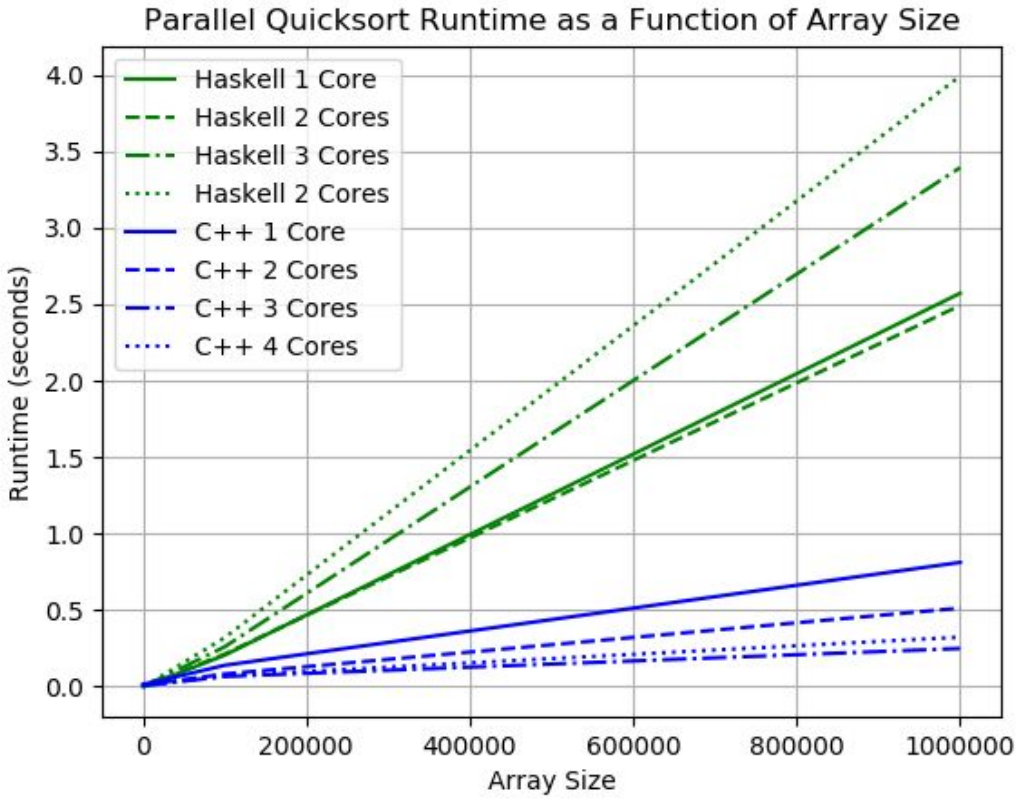
In contrast, all parallelism is defined explicitly with C++ MPI. This means every time a task is completed by one thread, it must send the data to another thread to continue execution of the program. Because the number of threads is defined before code

execution, and the code is executed by each thread individually, a master thread must be defined to divide the tasks and send them to the other threads. This causes the code to become very complicated very quickly, and the writability is significantly lower when writing parallel code compared to Haskell.

In terms of readability, our results were mixed. When we sent our code with functions and variables renamed, we received responses from five programmers. Of those five, three considered the C++ code to be more readable, while only two considered the Haskell code to be more readable. However, it should be noted that all three of those who indicated the C++ code to be more readable had prior experience with C++ and none with Haskell. The only programmer who responded who had no prior experience with either language chose the Haskell code as the more readable of the two. It should also be noted that none of those who responded were able to accurately determine what either of the functions did, other than to identify that the functions performed some actions in parallel.

Execution time

We decided not to compare the parallel sum algorithms with runtime, because the Haskell compiler will usually not run trivial operations such as addition in parallel, so we focused on comparing the quicksort algorithms on runtime. We took the average of several runs for each problem size, since quicksort's runtime can vary quite a bit based on the initial state of the array to be sorted.



The global sum algorithm yielded unexpected results. We expected to see that by adding more threads, the performance of the program would increase and terminate more rapidly. However, we found no increase in performance by adding more threads. Instead, the performance actually decreased. After obtaining this result, we looked more into why this might be occurring. We found that because there is so much overhead in Haskell when performing tasks in parallel, the cost of this overhead actually seems to outweigh parallel performance gains.

The global sum algorithm implemented in C++ MPI did work properly, and this is because parallel division of tasks is done explicitly. The program doesn't divide tasks as the need arises, but instead divides the tasks as defined by the programmer. As the program executed as intended, this showed us the stark difference between a functional language that divides tasks based on need and one which divides tasks explicitly as defined by the programmer.

The quicksort algorithm yielded more interesting results. With the data shown in the figure above, the Haskell quicksort only demonstrated slight improvement with two cores, while adding more caused too much overhead to improve performance. The C++ MPI quicksort accomplished the tasks in a significantly smaller amount of time, while showing that adding more cores does improve the speed of the algorithm.

Conclusions

Our conclusion is that Haskell's parallelism is easier to implement, but does not result in significant performance gains (and can even result in performance losses), while parallelism in C++ is more difficult, but does result in significant performance gains. One example of where Haskell's parallelism could be more desirable can be seen with Facebook's Sigma security system, which is designed to protect users from malware and other malicious internet activity. Facebook has indicated that one of the main reasons Haskell works well for Sigma's purposes is that its functional nature ensures that policies can be quickly implemented to run at the same time but cannot inadvertently interact with each other. However, for most purposes, the parallelism of C++ (and other imperative languages) is likely preferably, as the main reason parallelism is usually considered desirable is because of the runtime performance gains.