Haskell as an Introduction to Parallel Computing for Undergraduates

Barton, K., Dodson, S., Fenske, D., Whitehead, B., & Mache, J.
Department of Mathematical Sciences,
Lewis & Clark College, Portland, OR, USA

Abstract

As computing manufacturers increasingly rely on the use of multiple cores instead of improved clock speeds, programming in parallel has become an essential skill for computer scientists. Insufficient effort has been made to introduce the concept of parallel computing to the undergraduate curricula. We explored the pedagogical potential of a functional programming language, Haskell, in the context of parallel computing education. We extended previous work by providing both sequential and parallel Haskell implementations of a program that uses Riemann integration to calculate π.

Our assessment is that the parallel tools of Haskell are both concise and scalable. However, they can be difficult to learn, which brings us to our one reservation about using Haskell as a parallel programming language at an undergraduate level: learning functional programming may distract students from the original focus of learning general parallel programming concepts. As such, we expect that a module on parallel Haskell programming would be more effective if it was introduced in a programming languages course, which would presumably have already introduced students to the basic concepts of functional programming.

1 Introduction

Over the course of the last decade, advancement in computer processor speed has noticeably slowed. As circuitry speed begins to run up against the speed of light, it becomes increasingly clear that Moore's Law cannot be maintained with a single-CPU computing paradigm. Engineers have mitigated the slowing rate of increasing processor speed by building computing systems that include more than one core processor. Today, modern personal computers have anywhere from two to eight CPUs, and industrial-grade servers include processing blades with dozens. However, multiple core systems cannot deliver adequate improvements in processing speed without software support. Programmers must alter traditional, singular coding methods to properly take advantage of multiple cores. This is done by splitting process workloads into independent tasks

that can be accomplished in parallel. However, while good practices in parallel programming are vital to improving processing power, very little attention is paid to the discipline at the undergraduate level [1, 5]. Given the fact that parallel computing is a relatively new practice, no one language has emerged as the preferred parallel environment. Therefore this summer, we explored various parallel programming languages with respect to their pedagogical value for undergraduate computer science students[1]. Instead of focusing on one language that may become obscure in the coming years, it is more advantageous for students to learn a variety of languages, with the goal of understanding the benefits and downsides of each, while adopting a "parallel mindset" [6]. We searched for languages that were powerful, accessible, and informative about the nature of parallel programming in todays world of software engineering.

1.1 Haskell

Haskell was conceived as early as the 1980's, and a stable version was released in 2003 [2]. It is a purely functional language with a static, strong type system that incorporates automatic type inference. Haskell uses "lazy" evaluation in addition to optimized tail-call recursion in order to make recursive calls less resource-consuming. It also has built-in, convenient parallel interfaces which lead us to ask the question: can a functional programming paradigm be an effective environment in which to teach undergraduate students parallel computing?

When considering this question, educators must weigh the functionality and scalability of the parallel tools in Haskell against the overhead and possible distraction of teaching young programmers a new type of programming language structure. We decided to explore Haskell in parallel by implementing some simple, embarrassingly parallel problems demonstrated in the ACM Parallel Computing TechPack [7], namely, using Riemann integration of the unit circle in order to estimate π. We implemented this program with an eye for both scalability and teachability. Ideally, a parallel language strikes a balance such that its tools are powerful enough to run more complex systems in parallel, but are still simple enough to teach to undergraduate computer science students.

1.2 Parallel Computing

For more computationally expensive problems, a speedup can be achieved by dividing the solution into pieces that can be evaluated on multiple cores at once. While automatic parallelization is still not possible, Haskell is able to remove some of the annoying problems associated with traditional parallel computing techniques.

Haskell programs are deterministic, meaning that they always produce the same answer, so parallel programs can be run, tested, and debugged on a single core. Additionally, the programmer does not have to worry about classic problems like race conditions or deadlocks that are hard to detect and debug [3]. The low-level details of parallelism are abstracted in Haskell, so programs are

more likely to run on a range of hardware. Additionally, Haskell uses advanced runtime systems, like parallel garbage collection[3].

The main challenge with programming in parallel is finding the optimal partition of the problem. In other words, the problem needs to be divided into pieces that can be computed in parallel. There should be enough pieces that each processor or core is always busy [3]. This process of optimization can be achieved through a trial and error process.

2 Quicksort

As an introduction to the Haskell programming language, below are sequential and parallel implementations of the Quicksort algorithm [8]. For the readers not familiar with this sorting algorithm, a description of it follows. The algorithm sorts a list of integers. An element from the list is chosen to be the pivot value. A sublist of all the elements less than the pivot is created, and recursively sorted. A sublist of all the elements greater than or equal to the pivot is also created, and recursively sorted. Then the two sorted sublists are appended.

2.1 Sequential Quicksort

The sequential version of Quicksort is seen as one of the best examples of the elegance of the Haskell programming language, because implementations of the Quicksort algorithm in other programming languages, such as C, C++ or Java, are usually much longer.

Starting at line 2, the code sequentially executes, going through each case until it reaches one that applies to the given situation. The first guard, on line 2, describes the case of a non-empty list. The last case, on line 5, checks for an empty list, in which case the empty list is returned, because it is already sorted. If the set is nonempty, our sorted array will be found by recursively calling sort on the smaller and larger lists.

The `where` keyword, on line 3, is placed directly after a statement that needs to use the definitions stated in the where block. A `where` block provides local definitions so that there is no danger of ambiguity or reusing names in other parts of the program. On line 3 the set of all `xs` in the set which are less than or equal to the pivot is sorted recursively until we reach the case of an empty set, and then we retrace our steps through the recursion and concatenate the smaller lists to the larger lists. We do the recursion and evaluation in this order because Haskell is a lazy language, and will not evaluate statements until it is absolutely necessary. Haskell will save a list of functions to evaluate and not until it reaches the end of the recursive loop will it finally go back and find the outcomes resulting from each pass.

```
1 sort :: (Ord a) => [a] -> [a]
2 sort (x:xs) = smaller ++ x:larger
3     where smaller   = sort [y | y <- xs, y <  x]
```

```
4           larger    = sort [y | y <- xs, y >= x]
5 sort _ = []
```

2.2 Parallel Quicksort

With the addition of three functions the sequential Quicksort can be written in parallel. The main difference is that the code takes advantage of the `par` and `pseq` functions, which can be found in the `Control.Parallel` modules. The `par` function takes the argument to its left and begins working on it in parallel, while moving on to the right argument. The `pseq` function is almost the opposite of `par`. When we call `pseq`, the function requires that its left argument finish before it allows the right argument to execute. The force function simply prevents lazy evaluation, so that the left argument is evaluated eagerly.

```
1   import Control.Parallel
2 parSort :: (Ord a) => [a] -> [a]
3 parSort (x:xs) = force larger `par` (force smaller `pseq` (smaller ++ x:larger))
4     where smaller   = parSort [y | y <- xs, y <  x]
5           larger    = parSort [y | y <- xs, y >= x]
6 parSort _ = []
7 force :: [a] -> ()
8 force xs = go xs `pseq` ()
9     where go (_:xs) = go xs
10           go [] = 1
```

3 Riemann Integral

For demonstrating some of the features of Haskell, a professor could show a simple program that will estimate π by approximating the area under half of the unit circle and multiplying it by two. This has been a popular introduction to parallelism and has been used in the ACM Parallel Computing TechPack [7]. Our code could be a nice addition to this previous educational work, which has focused on more traditional parallel programming languages.

These programs are good class examples because they utilize the list features of Haskell, they are easy to relate to, and the transition to parallel code is fairly straightforward.

3.1 Sequential Riemann

The program will take an argument for the number of partitions and return an estimation of π. It will do this by the method of right-handed Riemann rectangle summation. To implement this sum we do the following. First we create a list that has an appropriate dx based on the number of partitions the user inputs. We then multiply dx by twice the height of the right hand point to get the area of our rectangle. We then add up all of the area of the rectangles to get our approximation.

```
1 -- Equation for the upper hemisphere of the unit circle
2 circle :: Double -> Double
3 circle x = sqrt (abs(1 - x^2))
4 -- Calculate the area of a right-handed Riemann rectangle
5 area :: Double -> Double -> Double
6 area x1 x2 = (x2 - x1) * circle x2
7 -- Recursively add the areas of the Riemann rectangles
8 estimate (x:[])     = 0
9 estimate (x:y:xs)   =  (area x y) + estimate (y:xs)
```

3.2 Parallel Riemann

The parallel version is almost identical code, using a similar recursive function to add the areas of the Riemann rectangles. The primary difference comes from the insertion of the `par` and `pseq` functions. In our parallel estimation of π, `par` is calculating smaller, and `pseq` is calculating larger at the same time. Larger makes a recursive call to `parEstimate`, giving us another smaller section to begin executing in parallel. This essentially gives us a cascading sum of parallel computations of the areas of the Riemann rectangles. Once larger –with the recursive smallers– is finally complete, larger and smaller are added together, Resulting in π.

```
1 import Control.Parallel
2 -- Equation for the upper hemisphere of the unit circle
3 circle :: Double -> Double
4 circle x = sqrt (abs(1 - x^2))
5 -- Calculate the area of a right-handed Riemann rectangle
6 area :: Double -> Double -> Double
7 area x1 x2 = (x2 - x1) * circle x2
8 -- Recursively add the areas of the Riemann rectangles
9   parEstimate :: [Double] -> Double
10 parEstimate (x:[]) = 0
11 parEstimate (x:y:[]) = area x y
12 parEstimate (x:y:xs) =
13 smaller `par` (larger `pseq` smaller + larger)
14      where   smaller = area x y
15              larger = parEstimate (y:xs)
```

3.3 Long-term Assignments

The Riemann integral is a nice introduction to Haskell and the power and simplicity of its parallel computing packages, but the ideas could be further explored in a long-term homework assignment [6]. There are a wide variety of embarrassingly parallel programs that could be applied using the Haskell tools described here. Professors could assign as long term projects various forms of population simulations, such as the Game of Life. These would be excellent, challenging

problems for young programmers that would introduce them to slightly more complex Haskell code. In the case of such an assignment, it might be beneficial to provide to students easy, intuitive graphics packages, so that students could focus more on the parallel, back-end aspect of their codes, rather than the challenge of creating good visualizations.

4 Discussion

In conclusion, we have found that Haskell is a programming language that can be used effectively in order to teach undergraduates about parallel computing. The parallel patterns in Haskell, like `par` and `pseq`, are straightforward and should therefore take little time to introduce to students. The fact that the number of cores to be used in the program can be specified when compiling means that parallel computing is scalable.

Our parallel π estimate, for example, should be able to take advantage of massively parallel threading systems, while also working on any personal computer. These properties make the parallel options in Haskell both easy to learn and powerful to use. There is, however, some question as to whether the language should be included in the curriculum of a course dedicated to parallel computing. While functional programming is compelling and powerful, it is also very different from any of the imperative languages that most undergraduates are familiar with. As such, the learning curve with Haskell can be steep, and may distract students from the focus of learning good practices in parallel computing. One solution could be to include a module on parallel computing with Haskell in a programming language structures course, rather than in a course dedicated solely to parallel computing. Therefore if students already understand functional programming, they can be focused on learning the parallel part of the code, deriving the maximum educational benefit from the work shown here.

5 Future Work

We would like to explore more challenging parallel problems. One interest is in Boids simulations which are used to simulate the flocking of birds [4]. This is an embarrassingly parallel problem. It may be a good introduction to parallel computing, because the visualization of the algorithm is attractive.

The simulation makes parallelization accessible to students because the idea that each thread is an object, or boid, on the screen will ultimately lead to better understanding of the theory as a whole. This simulation would introduce students to graphics in Haskell, while still keeping the fundamentals of the project relatively simple. This style of project has been shown to be particularly useful in educating students about parallel programming, because they will not get hung up on details of the code, but instead be able to grasp the overall ideas and advantages of parallelism.

We would also like to implement this tutorial in a programming languages and parallel computing class at Lewis & Clark College in the fall of 2013 and spring of 2014, respectively.

6 Acknowledgments

This research was funded by the John S. Rogers Science Research Program at Lewis & Clark College and the National Science Foundation (DUE 1044932).

References

[1] Joel Adams, Richard Brown, and Elizabeth Shoop. Patterns and exemplars: Compelling strategies for teaching parallel and distributed computing to cs undergraduates. In *EduPar-13*, 2013.

[2] Miran Lipovača. *Learn you a Haskell for great good! a beginner's guide.* No Starch Press, San Francisco, CA, 2011.

[3] Simon Marlow. *Parallel and Concurrent Programming in Haskell.* O'reilly & Associates Inc, Sebastopol, CA, 2013.

[4] Craig Reynolds. Flocks, herds and schools: A distributed behavioral model. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, 1987.

[5] Wilson Rivera. How to introduce parallelism into programming languages courses. In *EduPar-13*, 2013.

[6] Suzanne Rivoire. A breadth-first course in multicore and manycore programming. In *Proceedings of the 41st ACM technical symposium on Computer science education*, 2010.

[7] Paul Steinberg and Matthew Wolf. ACM Parallel Computing TechPack. `http://techpack.acm.org/parallel/`. Accessed: 2013-05-30.

[8] Bryan Sullivan. *Real world Haskell.* O'reilly & Associates Inc, Sebastopol, CA, 2009.