ADDING PARALLEL HASKELL TO THE UNDERGRADUATE PROGRAMMING LANGUAGE COURSE

David P. Bunde
Department of Computer Science
Knox College
dbunde@knox.edu

Jens Mache, Peter Drake
Department of Mathematical Sciences
Lewis & Clark College
{jmache, drake}@lclark.edu

ABSTRACT

Parallel computing is a new knowledge area in the 2013 ACM-IEEE curricular recommendations. This paper describes our experiences adding a brief module on parallel Haskell to the undergraduate programming languages courses at two colleges. We offer suggestions for others wishing to introduce parallelism in such courses.

Keywords

Parallel computing, programming languages course, Haskell, computer science education

1. INTRODUCTION

In recent years, multi-core CPUs have become all but ubiquitous at every level of computing, from supercomputers to mobile phones. Programmers trained only in traditional, serial programming are unable to take advantage of this parallel hardware. The 2013 ACM-IEEE curricular recommendations introduce parallel programming as a new knowledge area [4]. This paper describes our experiences adding parallel Haskell to the undergraduate programming languages courses at two colleges.

In the past, parallel programming education has often focused on explicit threading, MPI or OpenMP [10]. Some curricula offer a whole course on parallelism. Others scatter the material across other courses such as Computer Organization [2].

We found Haskell to be particularly suited to introducing and demonstrating parallel programming for two reasons. First, it is a functional language, meaning that most parts of the language are devoid of side effects, freeing the programmer from race conditions and other problems that plague parallel software written in traditional languages. Second, where it does provide mutable variables and other features that allow side effects, they are presented via higher-level constructs that give the programmer more expressivity and safety than explicit locks and threading.

In this paper, our focus is on (1) adding parallelism to the undergraduate programming language course and (2) the high-level parallel language constructs of Haskell.

2. PARALLEL QUICKSORT IN HASKELL

Before describing our experiences, we briefly survey the relevant features of Haskell. We start with the features related to functional programming.

The Control.Parallel module in Haskell includes the functions par and pseq, which allow a serial program to be easily parallelized. Each of these functions takes two tasks (represented as expressions to be evaluated), evaluates each of them, and returns the second one. The difference between the functions is when the tasks are evaluated; par requests simultaneous execution while pseq guarantees that the first task is evaluated before the second. par and pseq are often used together to parallelize code.

We illustrate these functions using quicksort, following the treatment of Sullivan et al. [9]. Haskell can express the serial quicksort algorithm in a remarkably concise way:

```
1 quicksort [] = []                          -- base case: the empty list is already sorted
2 quicksort (x:xs) = smaller ++ x:larger     -- otherwise, x is pivot
3     where
4       smaller = quicksort [y | y <- xs, y < x]  -- i.e. recursively sort smaller elements
5       larger = quicksort [y | y <- xs, y >= x]
```

Line 1 gives the base case: an empty list is already sorted. Line 2 defines the recursive case for a list with a head (x) and tail (xs). The result is the concatenation of smaller, the pivot x, and larger, with smaller and larger defined on the remaining lines. Line 4 defines smaller as the (recursively) sorted list of elements from xs that are less than x. Line 5 defines larger analogously.

This function is quickly parallelized by adding the par and pseq functions:

```
1 import Control.Parallel     -- required to use par and pseq
2 parQuicksort [] = []
3 parQuicksort (x:xs) = (($! smaller) `par` ($! larger)) `pseq` (smaller ++ x:larger)
4     where
5       smaller = parQuicksort [y | y <- xs, y < x]
6       larger = parQuicksort [y | y <- xs, y >= x]
```

The idea of parallelizing quicksort is not very complicated: make the recursive calls in parallel. The backquotes around par and pseq allow them to be used as inline operators. The overall effect is that smaller and larger are evaluated in parallel; pseq guarantees that both are evaluated *before* the append (++) operation. (This code is parenthesized incorrectly in Real World Haskell [9]; the version there produces the correct answer, but gains no speedup from parallelism.) The $! notation disables lazy evaluation of its arguments; without it the sublists are created only when their elements are read, effectively serializing the entire function.

Whereas the traditional (serial) code runs on only one core, the parallel code can utilize all available cores. With appropriate command-line arguments, the parallel version runs faster on multi-core CPUs, from supercomputers to mobile phones.

3. PARALLISM IN HASKELL WITH MUTABLE VARIABLES

We now discuss some of the concurrency-management features of Haskell that exist outside the functional programming paradigm. It may seem strange to discuss these features after talking about the advantages of functional programming for parallelism, but we do so for two reasons. First, our goal is not to teach Haskell per se, but to use it to illustrate programming language concepts, including different ways to manage concurrency. Second, these features have particularly nice realizations in Haskell due to its sophisticated type system.

3.1 MVars

The first non-functional-programming feature we discuss is MVars, mutable memory locations that can be shared between threads. What distinguishes these variables from standard shared memory global variables is that they have not only a value, but also a state of empty or full. An empty variable can be written to (making it full), but attempting to read from it causes the reading thread to block. Similarly, a full variable can be read from (making it empty), but a thread attempting to write to it will block. An MVar is effectively a blocking queue of length 1.

One nice use of MVars is to prevent a certain kind of race condition. Consider a program in which multiple threads each increment a shared counter. This type of operation is fairly common, which has led many languages to provide special operators for it (++, --, +=, *=, …). Performing any of these operations on a shared variable requires two memory operations, one to read the current value and then a second one to write the new value. A lost update error occurs when two threads each read the current value before either writes its new value. These threads will each write a new value reflecting only one of the updates, effectively meaning that only one update takes effect. If the shared value is an MVar, however, this type of bug cannot occur because the second thread's read will block until the first thread writes its updated value.

Thus, MVars are a useful abstraction that is slightly higher level than a lock (which they can be used to create). They are nice to show students because they are distinct from common concurrency primitives such as locks and semaphores while not being strictly tied to Haskell; they are called sync variables in Chapel [3].

3.2 Transactional Memory

The second non-functional-programming feature we discuss is transactional memory. The terminology comes from databases, where a transaction is a group of database operations performed atomically. Transactional memory allows a programmer to identify a critical section as a transaction and the runtime system guarantees that other threads will perceive the entire transaction as happening at the same instant. This has the potential of easing the additional burden of parallel programming by transferring some of it to the compiler and runtime system. Transactions are the subject of active research. Much of this research for one implementation approach, Software Transactional Memory (STM), is being conducted in Haskell.

Of particular interest for this paper is the work of Harris et al. [7]. This article explains recent research implementing (in Haskell) a system supporting transaction composition. Their motivating example is moving an entry between a pair of hash tables, composing the remove and insert methods. Traditional thread safety requires synchronization on each of these methods

separately, but explicit synchronization for the combined operation is needed to prevent exposure of the intermediate state when the value has been removed from one table but not yet added to the other. Lock-based solutions seem to require either exposing the locks in the hash table API (breaking the hash table abstraction and creating the potential for deadlocks or race conditions if the locks are misused) or explicitly modifying the hash table to support this operation (again, breaking the abstraction by requiring the hash table to anticipate all the ways it could be used).

The solution pursued in [7] builds on the Haskell type system, particularly the treatment of I/O. I/O is a particular challenge for functional languages since it cannot be purely functional; neither printing nor reading input can be repeated since either action changes the state of the world. Haskell's solution is for the type system to distinguish between functions that perform I/O (called IO actions) and those that do not. IO actions can be combined and passed around, but they do not perform I/O until invoked explicitly with special notation. Note that reading or writing to an MVar is an IO action.

Harris et al. [7] create a new kind of action (an STM action) to represent a transaction. The implementation is based on optimistic speculation; the transaction begins executing without setting locks, with support for being "rolled back" and all its effects undone if it is found to conflict with other transactions. (Note that this can only occur while the transaction is in progress; if no conflict occurs by transaction completion, it will never be rolled back.) Composition then comes from combining these actions just as IO actions can be combined. This approach uses the Haskell type system to segregate STM actions from IO actions since the side effects of IO actions mean they cannot be rolled back. The type system also segregates transactional code from purely functional code. This ensures that transaction variables cannot be modified outside of transactions, avoiding a flaw with some transactional memory systems, which rely on programming convention to prevent unprotected variable accesses.

As with MVars, transactional memory is a good topic for programming languages because it is a higher-level concurrency-control operation. As an important testbed for transactional memory research, Haskell is particularly suitable for demonstrating it.

4. CLASSROOM EXPERIENCE AT TWO COLLEGES

We now describe how we taught parallel aspects of Haskell at each of our institutions.

4.1 Lewis & Clark College

At Lewis & Clark College, in the fall semester of 2013, a parallel Haskell module was presented in two 60-minutes class periods of the Programming Language Structures course. At this point, the students had spent approximately three weeks working through the first 10 chapters of Lipovača [8]. The text was augmented by a series of YouTube videos [5] created to accompany the text.

Three undergraduate students who had explored parallel Haskell as part of a summer research experience (who were also students in the class) gave a presentation on the need for parallel computing, the benefits of using Haskell for such tasks, and introductory details on writing parallel Haskell code. The example was quicksort as described in Section 2. The presenters did a good job recovering from a technical glitch where some of the necessary

libraries had not been installed on the lab computers. (Another student had a laptop with a proper installation.)

After the presentation, students were asked to write a parallel program that estimates pi numerically by approximating the area under half of the unit circle (and multiplying it by two). This Riemann sum problem is a popular introduction to parallelism and has been used in the ACM Parallel Computing TechPack [1].

While the presentation was well-received, there was a noticeable "deer in the headlights" reaction when the class was asked to implement the Riemann sum algorithm in parallel. It is not clear how much of the lack of understanding came from the problem itself, how much from Haskell in general, and how much from the new parallel structures. This could probably have been ameliorated by breaking the exercise down into smaller steps.

A speedup was demonstrated for the quicksort problem, but not for the Riemann sum problem. It would have been more motivating to provide a demonstrable speedup earlier in the presentation and to let the students in the audience see a tangible benefit from their own work.

On an exam several weeks later, students were asked to explain the difference between par and pseq. Most (52%) of the students answered this question correctly; all of the others had approximately correct answers.

4.2 Knox College

At the Knox College, the parallelism unit was allocated two 70-minute lectures. (It actually got 10-15 minutes of the previous day as well; this time is included in the "first day" of the unit as described below.) The students had all previously seen the argument for the shift to parallel computing so the focus of this time was on seeing how parallelism could be expressed. For prior background, the students had spent approximately half a term on programming languages. This included 4 homework assignments of Haskell programming using the purely functional features of the language; giving students experience with functional programming was one of the course's main goals. Since the course is an upper-division elective, all the students had previously been exposed to the need for parallel programming and approaches using explicit threading.

The first day of the unit began with discussing the parallel sorting code, beginning with a serial implementation. Next came the introduction of par as a hint to the runtime system that two operations could be performed in parallel, with the observation that the operations could not interfere with each other because of the functional nature of Haskell. Next, pseq was presented as the way to represent a dependency. Once the students seemed comfortable with the basic code, the instructor told them that it was not actually parallel without the $! operators to force evaluation. This involved some review of lazy evaluation.

The day concluded with the presentation of MVar variables, relating them to the prior discussion of IO actions, and a brief introduction to the idea of transactional memory, motivated by describing it as a higher-level primitive for shared memory programming than the locks that they had used in previous classes. Due to the limited time, all of this material was covered in a

lecture setting so that students saw the code, but did not have an opportunity to play with it. The motivating example for both MVars and transactions was clearing checks (withdrawing from one bank account and depositing into another; clearing multiple checks concurrently easily creates a race condition).

Prior to the second day, the students were told to read the first three sections of Harris et al. [7] and the second day was spent discussing it. Since the students had limited prior experience reading technical papers, most of the time was spent on going through the paper paragraph by paragraph, making sure that the students could explain the key arguments. This seemed to be necessary since the students initially had difficulty doing so, but they seemed to understand the covered parts of the paper by the end of the period.

5. EVALUATION

To evaluate these modules, students at both colleges were given a brief survey. These surveys were designed to make the results as comparable as possible between the two colleges. The students at Knox were asked separately about the coverage of parallelism ("Parallel Haskell") and the transactional memory discussion ("Transactional memory"). The Parallel Haskell part is analogous to the module taught at Lewis & Clark except that it was shorter and also included MVar variables. The same questions were used except for being rephrased for the Knox students to remove references to "the lab" and removing a question about how long students spent on the lab; the module used there did not include a lab.

Most questions on the survey were multiple-choice questions on a 5-point Likert scale. Student answers on these questions are summarized below.

	Lewis & Clark College (n=21)	Knox College (n=7)	
		Parallel Haskell	Transactional memory
1. What was your level of interest in this topic? (1= "Very low", 5= "Very high"	3.33	3.57	3.57
2. What was the level of difficulty of this topic? (1="Very low", 5="Very high")	3.33	3.43	3.57
3. Approximately, how many hours did you spend on the lab? (1="≤2 hours", 2="≤4 hours", 3="≤6 hours", 4="≤8 hours", 5=">8 hours")	1.24	Not asked at Knox, which did not have a lab component.	
4. How much do you agree with the following? (1= "Strongly disagree", 5="Strongly agree")			
4a. The class time spent on this topic was worthwhile.	3.81	4.00	3.86
4b. This material contributed to my overall understanding of the other material in the course.	3.52	3.29	3.86
4c. I was sufficiently prepared to understand this material.	3.52	4.00	3.71
4d. As a result of this material, I am more interested in this topic.	3.71	3.29	3.71

Comparing the survey results between the two schools, the results are generally similar. The questions about the unit being worthwhile, contributing to their knowledge of material in the course, and increasing their interest in the topic, nearly all the ratings were either 3 or 4, indicating a neutral to slightly positive view. The largest difference in scores between colleges was in the question about the unit increasing interest in the topic; the Lewis & Clark average for

this question was 3.71 while the Knox average was only 3.29 (looking just at the "Parallel Haskell" rating). This suggests that the students found the longer unit more complete.

Based on a comparison of the Knox College scores of the two part of the unit, student impressions of the two parts were similar, but they felt that the transactional material was slightly harder and more interesting. The biggest difference in scores between the parts was that they felt that the transactional memory part of the unit gave a much greater contribution to the rest of the course. Since a significant part of the transactional memory paper concerns IO actions and this was given relatively light coverage due to the course's focus on Haskell's purely functional aspects, students may have appreciated another explanation and justification for this part of the language.

Looking at the free response questions, spending more time was the most common suggestion for improvement from students at both schools. At Knox College, six of the seven students asked for either more time or the addition of a programming assignment. We agree with the students and plan to spend more time in the future.

The question about the most important thing the students learned highlights the difference between the two groups of students. At Knox, where the course was an upper-level elective, the comments talk about Haskell and its primitives being different than approaches they had seen previously in other courses. At Lewis & Clark, the comments indicate that the unit was some students' introduction to parallelism, with comments like "Up til this point, I hadn't even known parallel programming existed!" and "That parallelism isn't as difficult as I thought." These responses suggest that the module is, at least with some students, achieving its goal of making parallel programming accessible to more students.

6. CONCLUSIONS AND FUTURE WORK

This paper has described our experiences of adding parallel Haskell to the undergraduate programming languages courses at two liberal arts colleges. These modules were moderately successful in introducing students to parallelism in general and parallel Haskell in particular. Overall, we felt that Haskell showed great promise as a way of introducing parallelism and demonstrating some higher-level constructs for managing it.

While the modules discussed in this paper take up less than a week of class time, they do require that students have some previous experience with Haskell. The more advanced version involving MVar also requires experience with monads.

For the Riemann sum example, achieving speedup proved difficult because using a recursive par function can create a new thread for every Riemann rectangle calculated. The resulting overhead can outweigh the performance gains from the parallel execution. In the end, this code was not able to outperform the sequential version of the program. If improved run times are desired, we recommend the parListChunk function, a special abstraction in the Control.Parallel module that allows the programmer to apply a map to a list by dividing the work into parallel "chunks" of data with a specified size [12]. This implementation allows for more specific performance tuning based on the number of cores the code is being run on. That said, it would be better to use an example where a speedup can be demonstrated without resorting to such advanced techniques.

We would also like to use more engaging and motivating examples. One reasonable possibility is minimax search in a game [11]. While alpha-beta-pruning causes complications, raw minimax search is a "pleasantly parallel" task with no interaction between the branches. For a game like *Othello* or *Connect Four*, parallelization should either speed up a complete search to some fixed depth or search more deeply in the same time; students would recognize either of these as an improvement. To allow the students to focus on the task at hand, we would provide most of the code for the rules of the game, as in the Learn Java in *N* Games assignments [6]. For simple games, this can be done with a fairly small amount of Haskell code; [5] provides a serial implementation of Tic-Tac-Toe with minimax search.

ACKNOWLEDGMENTS

This work is partially supported by NSF grants DUE-1044299 and DUE-1044932.

REFERENCES

[1] ACM TechPack - Parallel Computing, http://techpack.acm.org/parallel/, http://www.eapf.net/resources, retrieved 4/21/14.
[2] Bunde, D., Karavanic, K., Mache, J., Mitchell, C., Adding GPU Computing to Computer Organization Courses, Proceedings of the 3rd NSF/TCPP Workshop on Parallel and Distributed Computing Education (EduPar), 2013.
[3] Chapel language, http://chapel.cray.com, retrieved 4/21/14.
[4] Computer Science Curricula 2013, http://cs2013.org, retrieved 4/21/14.
[5] Drake, P., Haskell, http://www.youtube.com/watch?v=OfxCm_OarIg, retrieved 4/21/14.
[6] Drake, P., Sung, K., Teaching Introductory Programming with Popular Board Games, Proceedings of the 42nd ACM Technical Symposium on Computer Science Education (SIGCSE), 2011.
[7] Harris, T., Marlow, S., Jones, S., Herlihy, M., Composable Memory Transactions, CACM, 51 (8), 91-100, 2008.
[8] Lipovača, M., *Learn You a Haskell for Great Good*, No Starch Press, 2011.
[9] O'Sullivan, B., Goerzen, J., Stewart, D., *Real World Haskell*, O'Reilly, 2009.
[10] Pacheco, P., *An Introduction to Parallel Programming*, Morgan Kaufmann, 2011.
[11] Russel, S., Norvig, P., *Artificial Intelligence: A Modern Approach*, Prentice Hall, 2010.
[12] Totoo, P., Deligiannis, P., Loidl, H., Haskell vs. F# vs. Scala: A High-level Language Features and Parallelism Support Comparison, Proceedings of the 1st ACM SIGPLAN Workshop on Functional High-performance Computing, 2012.