Infusing Parallelism into Introductory Computer Science Curriculum using MapReduce

Matthew Johnson
Robert H. Liao
Alexander Rasmussen
Ramesh Sridharan
Dan Garcia
Brian K. Harvey

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2008-34
http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-34.html

April 10, 2008

Acknowledgement

We wish to thank Christophe Bisciglia at Google for the idea of using MapReduce in the introductory CS curriculum, and Google for funding the work of the student authors and the purchase of a cluster of PCs on which to run Hadoop. We would also like to thank Eric Fraser, Albert Goto, Mike Howard, and the rest of the instructional staff for allowing us to use the Millennium Cluster to develop our program. Finally, we would like to thank the developers of MapReduce, STk, Hadoop, and SISC.

Infusing Parallelism into Introductory Computer Science Curriculum using MapReduce

Matthew Johnson Robert H. Liao Alexander Rasmussen Ramesh Sridharan
Daniel D. Garcia Brian Harvey

Abstract

We have incorporated cluster computing fundamentals into the introductory computer science curriculum at UC Berkeley. For the first course, we have developed coursework and programming problems in Scheme centered around Google's MapReduce. To allow students only familiar with Scheme to write and run MapReduce programs, we designed a functional interface in Scheme and implemented software to allow tasks to be run in parallel on a cluster. The streamlined interface enables students to focus on programming to the essence of the MapReduce model and avoid the potentially cumbersome details in the MapReduce implementation, and so it delivers a clear pedagogical advantage.

The interface's simplicity and purely functional treatment allows students to tackle data-parallel problems after the first two-thirds of the first introductory course.

In this paper we describe the system implementation to interface our Scheme interpreter with a cluster running Hadoop (a Java-based MapReduce implementation). Our design can serve as a prototype for other such interfaces in educational environments that do not use Java and therefore cannot simply use Hadoop. We also outline the MapReduce exercises we have introduced to our introductory course, which allow students in an introductory programming class to begin to work with data-parallel programs and designs.

1 Introduction

Computer science today is rapidly moving towards parallelism as a means to surmount increasing problem sizes and the declining rate of clock speed improvements. Despite the changing environment, undergraduate curriculum, particularly at the introductory level, often provides little or no coverage of basic parallel programming concepts. Our project introduces parallelism to the lower-division computer science courses at UC Berkeley, primarily our very first course, CS61A. We address cluster computing instead of multithreaded parallelism because the details associated with multiprocessor systems would exceed the scope of the introductory courses, and many of the data-parallel programming concepts remain the same. There is an entire upper-division course dedicated to multi-core concurrency issues [9]. By covering parallelism in introductory courses, students gain an earlier exposure to and greater appreciation of its advantages, disadvantages, and uses.

In CS61A, we chose to teach cluster computing by introducing Google's MapReduce [2] at a high level of abstraction, as it provides both a practical and intellectually compelling example of a highly parallelized system. MapReduce is presented entirely through a Scheme interface which works with Hadoop (an open-source implementation of MapReduce) [8] to execute student programs on a new cluster. In this paper, we describe the Scheme-Hadoop interface and outline our implementation, as well as provide examples of the material we developed for the course.

2 About CS61A

UC Berkeley's introductory computer science course is based on Abelson and Sussman's *Structure and Interpretation of Computer Programs* [1]. Programming for the course is done entirely in Scheme, a dialect of Lisp, and the only previous programming experience required is familiarity with the concept of recursion. The course gives students a broad overview of computer science concepts including functional programming, higher-order functions, abstract data types, abstraction techniques, and language implementation.

Each week, the course of 200 to 400 students meets for two to three lectures and smaller 30-person lab-

oratory and discussion sections. In addition, the course assigns weekly homework and four programming projects spread over the course of the semester. The students run their programs in a modified version of STk [3], a Scheme interpreter written by Erick Gallesio and customized for use in UC Berkeley CS instruction.

3 MapReduce Basics

MapReduce [2] is a programming paradigm and framework developed by Google for simplified, parallel data processing on clusters of computers. As its name implies, it was inspired by the Lisp list-processing functions `map` and `reduce`.

In Scheme, `map` and `reduce` allow for convenient expression of functional programming operations:

```
(map (lambda (x) (* x x)) '(1 2 3 4))
⇒ (1 4 9 16)
(reduce + 0 '(1 4 9 16))
⇒ 30
```

The MapReduce process was inspired by a composition of the two functions:

```
(reduce + 0 (map (lambda (x) (* x x)) '(1 2
3 4)))
⇒ 30
```

However, the MapReduce model behaves slightly differently. Although MapReduce uses ideas from functional programming, its model is in fact sequential, not functional; the mapper and reducer are called once per process and loop through their input data. All input, output, and intermediate data are expressed as key/value pairs. Programmers must provide "mapper" and "reducer" procedures, analogous to the function arguments provided to `map` and `reduce` above. The MapReduce infrastructure partitions and distributes input key/value pair data across a cluster and executes instances of the mapper procedure on the input in parallel. Unlike the Lisp composition above, the mapper instances emit intermediate key/value pairs, which are then grouped and sorted by key. The group of values associated with each key is processed by an instance of the reducer procedure, producing output key/value pairs. For a more detailed explanation of MapReduce, see [2].

4 Scheme Interface Design

In designing the Scheme interface for MapReduce, our highest priority was simplicity. We did not want to expose students to any unnecessary details of the inner workings of Hadoop; we wanted the interface to be opaque beyond the MapReduce model in its purest form, so that students could express the required procedures in familiar terms. Despite the need for simplicity, we did not want to compromise MapReduce's expressive power.

Our final interface function takes the following form:

```
(mapreduce mapper reducer reducer-base input)
```

Here, the `mapper` is a function that acts on a single key-value pair and returns a (possibly empty) list of corresponding intermediate key-value pairs as output. The `reducer` is a Lisp-standard reducer function, with a next-value and value-so-far as inputs, and some combination of the two as output. When Hadoop finishes, our system presents a stream to the student with the MapReduce output.

To perform a distributed grep operation on files identified by the `text-ID` string, a student would enter the following expressions into STk:

```
(define (mapper doc-line-pair)
  (if (match? pattern (cdr doc-line-pair))
      (list doc-line-pair)
      nil))
(mapreduce mapper cons nil "text-ID")
```

Here, each key-value pair input to the `mapper` has a document name as its key and a line of text from the document as its value.

To perform a distributed word count, returning a table mapping words to their number of occurrences, a student would enter the following:

```
(define (mapper doc-line-pair)
  (map (lambda (wd) (cons wd 1))
       (cdr doc-line-pair)))
(mapreduce mapper + 0 "text-ID")
```

The inner map function is used to break the line-by-line input from the `text-ID` files into words.

Using a powerful high-level language such as Scheme allowed us to greatly simplify the students' interface to MapReduce. For example, in Hadoop's Java implementation of the distributed grep example above, the code required to initialize and carry out

the MapReduce task is more than 50 lines. Additionally, students do not need to learn anything about the structure of Hadoop to be able to implement parallel MapReduce tasks; the Hadoop-specific objects, framework, and configuration files are entirely avoided, allowing students to focus on learning the programming model and its uses. Our interface also matches the functional composition model that inspired MapReduce much more closely than the original interface.

5 Implementation in Hadoop

The Scheme interface required a system that would allow a student using STk to start a Hadoop instance on the cluster, transport the Scheme code and environment to the worker nodes (the "tasktrackers"), interpret the Scheme code in parallel map tasks and reduce tasks, and return the generated output to STk in a tractable form.

In this section we give an overview of our system design and discuss particular implementation challenges.

5.1 System Overview

For a summary diagram of the system, see Figure **??**.

There are two fundamental parts to the system: the front-end node and the "worker" nodes. The front-end node is used for user interaction, serial computations, and initiating MapReduce jobs on the cluster, and so it runs STk, a Java Virtual Machine (JVM) providing the Java Native Interface (JNI) [7], and the Hadoop jobtracker JVM. The worker nodes run the Hadoop tasktrackers that carry out the parallel Map and Reduce computations, and thus each runs an instance of a Java-based Scheme interpreter.

Users run MapReduce tasks and interact with output files entirely through STk on the front-end node. The first `mapreduce` function invocation in STk sets up a JNI JVM to access the Hadoop API. The `mapreduce` invocation in STk writes the user's Scheme code to the Hadoop Distributed File System (HDFS), along with the current environment so that the worker machines can make reference to the user's local variables, and uses the JNI JVM to initialize Hadoop and pass in strings to identify the code, environment, and input file locations on HDFS. Input files are large, predetermined datasets that exist on HDFS, and so users pass in an identifying string to the `mapreduce` invocation which is mapped to a predetermined HDFS path. After the parallel computation is carried out and output files are copied from HDFS to the front-end node's local filesystem, local output filenames are passed from the jobtracker JVM through the JNI JVM and back to STk, which presents these files to the user in the form of a single lazy-evaluated stream of key-value pairs.

On the worker nodes, Hadoop tasktracker instances execute Java-based Scheme interpreters as both the mapper and reducer classes. The interpreters load the appropriate Scheme environment along with the Scheme code from HDFS. Input data files are parsed line by line and presented with the document name as the key. The intermediate pairs are typed as extended Hadoop Text objects, and all Scheme values (including numbers) are emitted as Text objects. As output key/value pairs are emitted they are written to HDFS, and once the reductions finish they are copied to the front-end node and presented to STk.

5.2 Implementation Challenges

5.2.1 The C/Java Barrier and JNI

The Berkeley STk interpreter is implemented in C, while the Hadoop API is entirely in Java. To allow STk to communicate with the Hadoop JVMs, we employed the Java Native Interface (JNI). The first time the `mapreduce` function is invoked, a JVM is instantiated and the appropriate Hadoop classes are run (subsequent invocations use the same JVM instantiation). STk passes the necessary file identifier strings into the JNI JVM, and once the parallel computation is complete, output filename strings are returned to STk via JNI.

5.2.2 Exporting the Environment

Scheme manages variable-value mappings in constructs called environments. An environment consists of frames containing variable-value mappings, and frames hierarchically point to other frames for the purpose of scoping. During execution, Scheme looks up variables with respect to a frame, following the hierarchy of frames as necessary until the global (root) frame is reached.

For our system to operate correctly, STk must export the entire environment at the point of the `mapreduce` invocation to the mapper and reducer

JVMs so that the user's Scheme code can be executed on the cluster in the correct context.

Scheme variables can refer to a variety of types, including numbers, symbols, pairs, and closures. A pair is a simple abstract data type consisting of two elements. A closure represents a procedure, and each closure must be linked to the frame in which it was defined. These are among the most commonly used types in our course, and as such we chose to support their usage and export. This choice also meant that we do not support other, less commonly used datatypes, such as vectors (arrays).

Our environment export solution stems from the realization that the Scheme environment is a directed cyclic graph. Nodes in this graph are either frames, closures, or pairs, and edges represent their respective frame dependencies. To serialize the environment we simply encode it as an adjacency list, associating each frame, closure and pair with a unique identification number that is created when the object is initialized within STk. STk writes the serialized environment to HDFS, and the mapper and reducer Scheme interpreters load the environment accordingly.

5.2.3 Java-Based Scheme Interpreter

Hadoop's mapper and reducer classes must be able to interpret Scheme code in parallel. Towards this end, we wanted a Scheme interpreter that would easily interface with the Hadoop API, so we chose the Second Interpreter of Scheme Code (SISC) [5], a Java-based Scheme interpreter. Each instance of the Hadoop mapper and reducer classes runs an associated instance of the interpreter.

5.2.4 Cascading Jobs

More sophisticated MapReduce operations, and more useful and compelling examples, require MapReduce tasks to be cascaded such that output generated from one operation is used as input to another. Our design separates the local data with which students directly interact in STk from the data available to the cluster on HDFS, so that MapReduce output on HDFS must be copied to the local disk so it can be viewed in STk. To allow output data to be used again as input data, we leave the original copy on HDFS and return an identifying string to STk, so that if the student uses the output data in another call to the `mapreduce` function, the identifying string is passed back to Hadoop and used to find the appropriate data. This approach avoids a significant amount of communication between the local filesystem and HDFS which would be impossible to accomplish given the class size, but it also means that ordinary Scheme operations in STk cannot be applied to the data on the local machine in between MapReduce tasks. Since the input and output data are usually very large, serial operations are considered infeasible, and thus our approach does not preclude computations that would be useful in the class context.

6 MapReduce Curriculum in CS61A

The MapReduce material in CS61A consists of two lectures, one homework assignment, and one lab assignment. The first lecture occurs early in the semester, and introduces the MapReduce model in a non-parallel version that shows the use of key-value pairs and makes explicit the grouping and sorting by key that comes between the Map and the Reduce parts of a MapReduce invocation. The second lecture is late in the semester and focuses on parallelism and on the details of using the parallel cluster. Fundamental parallelism concepts are explained and the STk-Hadoop interface is introduced. Most of the students' time in the latter week is allotted to hands-on practice with the lab and homework, and so our curriculum development focused on creating problems for the students to solve with MapReduce.

The programming problems needed to be tractable for students with no previous coursework in computer science but also interesting and useful enough to justify the pragmatism of MapReduce. The starting point for our brainstorms was the original MapReduce paper and its provided examples. Below is a list of some of the problems given in the lab and homework.

1. **Word Count Table** — Given a large set of documents, construct a table mapping words found in the documents to the number of instances of the word. Then, using the word count table as input, write another MapReduce step to generate a list of words used only once.

   The word count problem is one of the examples supplied in the original paper, and is a good first step in understanding the MapReduce paradigm. The extension of finding words used only once exemplifies the potential usefulness of cascaded MapReduce computations.

2. **Inverted Index** — Given a large set of documents, construct an index mapping words to the list of documents in which they appear. As an extension, filter out words shorter than a given length, so as to avoid unimportant words such as "a," "so," and "the."

   The inverted word index is another of the original Google-supplied examples, and it provides a simple motivation for MapReduce's usefulness in the context of search engines and information retrieval. The word length filtering allows students to practice with a mapper that emits an empty list, a difference from Scheme's `map` behavior. We provide a subset of the documents in Project Gutenberg [6] as the input.

3. **Spam Filter** — We want to implement a simple spam filtering idea by producing a "blacklist" of e-mail addresses that are spamming users. Spam messages are sent to many addresses at once, and so a spam message can be identified by the most commonly used subject lines; if an address sends many messages with the most common subject lines, it is likely a spamming address. Given a large collection of (simulated) e-mail server records, find the top ten addresses that have sent the most messages with frequently-recurring subject lines.

   This problem is much more involved, since it requires three MapReduce steps to complete, but it is also more realistically useful. The first step is to generate a table with subject lines as keys and counts of occurrences as values. From the tabulation of subject lines, another MapReduce step is used to sort the table by occurrence counts, so that the most commonly-used subject lines can be identified. Finally, given the ten most frequent subject lines in a short "violation" list, another MapReduce step is used to make a table with from-addresses as keys and counts of "violation" e-mails sent as values. An additional step could be added to sort the final table by violation counts, but since it would be identical to the second step, this additional step is omitted.

4. **Conceptual Questions** — Could you use MapReduce to perform a parallelized Sieve of Eratosthenes? If so, describe the process briefly. If not, why not?

   This question gives an example of a problem that requires global state to be shared between parallel workers, and thus it does not fit naturally into MapReduce's functional, data-driven model.

We met two principal difficulties when brainstorming problems: a limitation on data structures presented in the introductory course, and an inherent redundancy among the MapReduce examples. The first difficulty exists simply because CS61A does not assume any previous familiarity with data structures such as graphs, and so several ideas involving graph traversal (motivated by map search) could not be used.

The second difficulty was a realization that each of the single-step MapReduce tasks that we could brainstorm could essentially be reduced to one of the MapReduce examples in the original paper. The original examples enumerate the useful variations of input/output behaviors very thoroughly, and so within the domain of simple mapper and reducer functions, many problems began to sound the same. It was this fundamental redundancy in simple MapReduce examples that motivated us to design multi-step problems, decomposing larger computations into the simple tasks that MapReduce handles best.

7 Parallelism in Other Courses

CS61A provides an overview of programming in a high-level language, and so its discussion of MapReduce involves only the high-level parallelism model. The other courses in the introductory series, CS61B and CS61C, have a lower level of abstraction in their approach to computer science, and so their treatment of cluster computing varies accordingly.

CS61B is a Java-based course on data structures and algorithms. It is also intended to give students experience working in larger, more complex programming projects. Thus it was natural for CS61B to expose students to the Java-based Hadoop system and allow them to program directly with the API. For CS61B we developed a homework assignment involving a hill climbing optimization problem; given a row of contiguous movie theater seats and a set of interpersonal relationships (enemy, friend, boyfriend, etc.) that specify neighbor preferences, find the best possible seating arrangement. The assignment introduces the basics of iterative optimization and parallelization over random initial conditions, as well as the structure of the Hadoop system, and so it is well-suited for the CS61B abstraction level.

CS61C is the lowest level course, introducing C and

assembly programming, operating systems concepts, and machine architecture. It is also the first course to discuss overhead issues associated with memory, disk, and network communications, and so the parallelism discussion in CS61C covers similar issues involved in optimizing parallel speedup. The CS61C curriculum we developed consists of a lecture on cluster parallelism, providing an overview of both MapReduce and the more flexible Message Passing Interface (MPI) and their respective tradeoffs, as well as a two-hour programming lab. The first incarnation of the lab exercise focused on teaching simple MPI programming techniques and subroutines in C, and students were able to execute their MPI code on one of the University's clusters. However, after hearing student feedback, it was clear that the lab should focus more on speedup issues and benchmarking for more realistic programs and less on learning MPI. The new lab is currently under development, and will be centered on a case study of an MPI version of Conway's Game of Life and how its parameters (board size, number of nodes and processes, geometry of decomposition, communication block size) affect the parallel speedup.

8 Related Work

The University of Washington has an upper division course based on MapReduce and distributed systems[4]. However, since their course is targeted at more advanced students, they spend significantly more time discussing the details of distributed computing and networks. Their course is also taught in Java, and as a result they use Hadoop directly. However, this involves significantly more pedagogical overhead than our approach; for example, they devote an entire lab to learning how to use Hadoop.

9 Summary and Future Work

The new Scheme interface allows CS61A students to write and execute real MapReduce programs while maintaining an abstraction level appropriate for the course, and the subsequent courses effectively "lift the hood."

We launched the new curriculum along with the STk-Hadoop interface during the Fall 2007 semester. The interface scaled well to the 30-student lab sections and all the students were able to work concurrently. However, we encountered several setbacks while running the labs. Since the machines and networks were new and relatively untested, there were several instances of communication failure between the lab machines and the cluster. Additionally, we found the performance to be lacking: fixed setup overhead time was about 30 seconds for any job, and Scheme programs performed over 10 times slower than equivalent Java programs. We believe the loss to be somewhere in the I/O handling of the cluster-side software, and we anticipate that we will be able to reduce the slowdown to an acceptable level.

As a result, students' responses were mixed. Most were excited about the new material while others were frustrated by the technical difficulties and occasional slowdowns. To address these problems in the immediate future, we plan to provide smaller instances of the exercise data sets so that the same MapReduce programming can be practiced off-line.

Our most significant future work is to tune both the cluster's setup and the STk-Hadoop interface to increase its reliability and performance. One area that is particularly ripe for optimization is the client-side Scheme interpreter's processing of input and output. Since Scheme has provided such a clear and simple way to express MapReduce programs, we also plan to investigate other modern programming paradigms that we may be able to incorporate into Scheme in a similarly clear, educational manner.

All of our curriculum content and Scheme bindings will be available as a free download from the section of the Google Code for Educators website specializing in Distributed Systems [4].

References

[1] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Massachusetts, 1999.

[2] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *OSDI*, 2004.

[3] E. Gallesio. STk. Website. http://kaolin.unice.fr/STk/.

[4] Google. Google Code For Educators, Distributed Systems. Website. http://code.google.com/edu/content/parallel.html.

[5] S. G. Miller. SISC - Second Interpreter of Scheme Code. Website. http://www.sisc-scheme.org/.

[6] Project Gutenberg. Project Gutenberg. Website. http://www.gutenberg.org/.

[7] Sun Microsystems Inc. Java Native Interface. Website. http://java.sun.com/j2se/1.4.2/docs/guide/jni/.

[8] The Apache Software Foundation. Hadoop. Website. http://lucene.apache.org/hadoop/.

[9] K. Yelick. Parallel Programming for Multicore. Website. http://www.cs.berkeley.edu/ yelick-/cs194f07/.