Simplifying the Development and Deployment of MapReduce Algorithms

FEROSH JACOB, AMBER WAGNER, PRATEEK BAHRI, SUSAN VRBSKY, AND JEFF GRAY
Department of Computer Science
University of Alabama

MapReduce algorithms can be difficult to write and test due to the accidental complexities involved with existing MapReduce implementations. Furthermore, the configuration details involved in running MapReduce algorithms within a cloud present a set of new challenges. Our research reveals that many details of cloud configuration can be hidden from programmers in an automated and transparent manner. Using concepts from software engineering, we have increased the ease of use for implementing MapReduce algorithms by creating a lightweight domain-specific language (DSL). Additionally, we created a plug-in for the Eclipse integrated development environment (IDE) based on this DSL to automate and hide many cloud configuration details. The goal of the combination of our IDE and DSL is to improve the efficiency and effectiveness for programmers in developing MapReduce algorithms for cloud computing.

This paper describes the existing challenges of creating MapReduce algorithms and how our approach minimizes these challenges. MapRedoop is a framework that can be used to transform a program written in a DSL to a MapReduce implementation, which can be deployed and executed in a cloud platform such as Eucalyptus or Amazon's Elastic Compute Cloud (EC2). Assorted examples selected from various domains have been rewritten in the MapRedoop framework to demonstrate its expressiveness and usefulness. Our performance analysis reveals that the advantages gained using our approach can be attained with comparable execution times to the methodologies currently in practice.

Keywords: MapReduce, Hadoop, DSL, Execution Environment.

1. INTRODUCTION

Cloud computing provides users with the flexibility of performing high volume computations without the cost of building the required infrastructure. There are several purposes for writing software within a cloud architecture, such as file storage and management [Ghemawat et al. 2003], cloud infrastructure management [Sugiki et al. 2010], and computations of large datasets [Manjunatha et al. 2011]. The focus of this paper is on writing MapReduce algorithms, which is a programming model used to solve problems involving large data sets utilizing a cluster of computation nodes where input and output are converted to key/value pairs [Dean and Ghemawat 2008]. The MapReduce model allows: 1) partitioning the problem into smaller sub-problems, 2) solving the sub-problems, and 3) combining the results from the smaller sub-problems to solve the original issue. The programming model automatically partitions the problems based on the input given (e.g., splitting the input into lines or blocks of lines if given a text file; files if given a directory; and objects if given a list of objects). MapReduce is responsible for solving the sub-problems in parallel and makes the individual results available for the combiner to act upon. From the programmer's perspective, MapReduce involves two main computations:

(1) **Map:** implements the computation logic for the sub-problem; and
(2) **Reduce:** implements the logic for combining the sub-problems to solve the larger problem.

According to [Dean and Ghemawat 2008], since its development, "more than ten thousand distinct MapReduce programs have been implemented internally at Google over the past four years, and an average of one hundred thousand MapReduce jobs are executed on Google's clusters every day, processing a total of more than twenty petabytes of data per day."

Although Google was among the first to implement and utilize MapReduce, Apache (Hadoop) and Stanford (Phoenix) have created open source implementations [Dean and Ghemawat 2008]. Google identified five areas for the refinement of MapReduce, including customized input and output types and simplified debugging [Dean and Ghemawat 2008]. Not all MapReduce problems will require the same input type (e.g., text file, associative array, clusters, vectors); therefore, if a programmer wishes to use a different input type, it is the programmer's responsibility to create a conversion class. Google's solution for customized input and output types is for the programmer to create a custom reader that will convert the input into the required key/value pairs for the mapper, which are called protocol buffers[1] (PB). PBs create data structures automatically, but the programmer must create the appropriate converters. Although this solution did improve Google's MapReduce implementation, the responsibility still lies with the programmer. When debugging a MapReduce algorithm in Google's original implementation, the programmer was forced to debug in the cloud at runtime. Google's solution for simplified debugging is to use a specified flag to run the code locally, at which time the programmer can use the debugging or testing tool of choice [Dean and Ghemawat 2004]. This gave the programmer the choice to test either in the cloud or locally.

Our solution to the accidental complexity of customized input/output that emerges in MapReduce solutions is to present a domain-specific language (DSL), which is an expressive language focused on a specific problem domain that provides a higher level of abstraction for specifying a computational or configuration need in the domain [Wu and Gray 2005; Raja and Lakshmanan 2010; Mernik et al. 2005]. According to [Mernik et al. 2005], an important benefit of a DSL is that a programmer's focus may lie on the problem space as the accidental complexities of the solution space are minimized. Additionally, Raja and Lakshmanan explain that a DSL should "screen away the internal complex operations of the system." It is desirable for the programmer to focus on the correctness of the algorithm and not be concerned with platform and configuration issues.

MapRedoop is our contribution that addresses the second accidental complexity related to simplified debugging of a MapReduce algorithm. MapRedoop is a framework consisting of a DSL and an integrated development environment (IDE) within Eclipse. In MapRedoop, the programmer is given the flexibility to implement the map and reduce functions after specifying some of the data structure details, such that the user is oblivious to the setup reuired to execute the program and any possible type mismatches that might occur. Because MapRedoop is a plug-in for Eclipse, a programmer may develop MapReduce algorithms within an IDE, in which the programmer has the option to execute the code running Hadoop either in a cloud or on a local machine instance. Section 5 describes how MapRedoop is used to address the challenges of implementing MapReduce algorithms.

This paper discusses our motivation for implementing MapRedoop in Section 2. We describe related work in Section 3 and provide additional details on MapReduce programs in Section 4. A discussion of the benefits of applying MapRedoop can be found in Section 5. An evaluation of MapRedoop on several examples is provided in Section 6. The final section of the paper includes a summary of future work and conclusions. A demo using MapRedoop programs can be found on our project site[2].

2. MOTIVATION

As a motivating need for the approach described in this paper, our first experience with writing a MapReduce algorithm was confusing and frustrating. We wanted to modify the WordCount[3] example algorithm to compute the probabilities of bigrams beginning with a specific word occurring within a given text file. The algorithm itself was straightforward, but the environment,

[1]http://code.google.com/apis/protocolbuffers/
[2]https://sites.google.com/site/mapredoop/
[3]http://wiki.apache.org/hadoop/WordCount

particularly testing, data types, and class interactions presented challenges to our implementation.

In our initial solution using the pseudocode in Algorithm I, we encountered several accidental complexities that contributed multiple challenges during the development process. The primary issue we experienced was not demonstrated until we ran the code: we received zero for each of the probabilities. We knew what the proper data type for each variable should be (int, double, float), but Hadoop did not accept any of these types. Hadoop required the use of internal data types; therefore, we had to change each instance of the `IntWritable` data type to the `FloatWritable` data type. After we corrected the class and attempted to execute the program, we received a type mismatch error ("Type mismatch in key from map"), and we realized we needed to alter the data type in additional classes (`Driver`).

Input - R: Text files and a word w Output - Bigram probability of the word in the files
mapper **if** word = w output ($bigram$, 1) output ($word$, 1) **endIf**
reducer **while** $morevalues$ $sum = sum + values$ output ($sum/total$)

Algorithm I: Bigram probability estimation

A secondary issue we encountered was that the output of the mapper must match the type of the input of the reducer. Additionally, if a partitioner or combiner were included in our program, the output of the mapper would have to match the input of the partitioner, the output of the partitioner would have to match the input of the combiner, and finally, the output of the combiner would have to match the input of the combiner. This is a simple issue to which a programmer acclimates after writing a few MapReduce programs; however, we feel this is yet another accidental complexity. The situation differs when reading data from a text file versus reading the text file itself; Hadoop reads text files slowly when compared to sequential files. Therefore, the data from a text file must be converted to data in a sequential file to decrease the program's run-time. This experience caused us to ask a few questions:

(1) How can the programmer easily identify the required input requirements?
(2) Should the programmer need to be concerned with data types?
    —Is there a way the programmer can use familiar data types and then use Hadoop to internally convert these data types appropriately?
(3) With what other issues should the programmer not have to be concerned?

We feel there are three primary areas about which the MapReduce programmer should not have to be concerned:

(1) **Input structure:** The current frameworks, which claim to address these issues, have not solved the issues entirely. For example, a K-means [Kanungo et al. 2002] program executed in Mahout[4] (a library for machine learning and data-mining programs) expects a vector as an input; however, if the input structure differs, the programmer has to rewrite the file to match the structure which Mahout supports.

[4]http://mahout.apache.org/

(2) **Improper level of abstraction:** Ideally, the programmer should have the ability to focus solely on the map and reduce functions rather than the implementation details. Currently, the MapReduce programmer has to search within the source code to identify the mapper and the reducer (and depending on the program, the partitioner and combiner). After identifying these classes, the programmer has to delve deeper into the code to determine the proper inputs. The key challenge is that there is no central place where the required input values for each of these classes can be identified in order to increase program comprehension.

(3) **Improper validation:** Because the input and output for each class (mapper, partitioner, combiner, and reducer) are declared separately, mistakes (such as the data type issue we mentioned previously) are not identified until the entire program is executed. The programmer should have the ability to execute each class separately for validation purposes.

Upon identifying these three primary issues, we built a tool to aid the programmer in creating MapReduce programs. Our tool and its implementation are described in Section 5.

3. RELATED WORK

There have been many efforts in providing DSLs to express computational intensive problems, specifically in the parallel programming domain [Charles et al. 2005; Diaconescu and Zima 2007; Allen et al. 2007; Jacob et al. 2010; Fritz et al. 2004; Chafi et al. 2011; Jacob et al. 2009]. Recently, there have been many studies about the usefulness of DSLs in Cloud Computing, including work by the following: [Manjunatha et al. 2011; Ranabahu et al. 2010; Low et al. 2010; Sugiki et al. 2010; Pike et al. 2005], McCullough[5], and Kromer[6]. The six other efforts that are most relevant to our work are briefly presented here.

Kumoi is an embedded DSL for virtual data center management. Their primary goal was to "provide maximum management efficiency for experienced administrators" [Sugiki et al. 2010]. Kumoi utilizes a DSL to allow data center administrators to write complex management scripts while hiding unnecessary details. This tool reduced the number of lines of code required to deploy VMs by 71%, balance VMs by 81%, and shutdown VMs by 98% as compared to the same scripts written in Libvirt (Java). This tool is similar to our MapRedoop in that it simplifies the code necessary for programmers to write; however, Kumoi relies on a distributed object model (Java RMI) rather than a parallel function model (MapReduce).

OptiML is a DSL from the machine learning domain. It automatically analyzes and optimizes the domain specification generating CUDA[7] code [Sujeeth et al. 2011]. According to Low et al. "GraphLab achieves a balance between low-level (PThreads) and high-level (MapReduce) abstractions" [Low et al. 2010]. The developers of GraphLab report a significant speedup among the various ML (Machine Learning) algorithms tested, and thus, the goal of balancing high-level and low-level abstractions while improving efficiency was met. MapRedoop differs from GraphLab in that MapRedoop is based on creating an abstraction for MapReduce (a high-level language according to Low et al [Low et al. 2010]).

Manjunatha et al. [Manjunatha et al. 2011] present a DSL, Metabolink Toolkit, for scientists to analyze Nuclear Magnetic Resonance based metabolomics data. While the Metabolink Toolkit can be implemented on multiple platforms including Apache's Hadoop (taking advantage of MapReduce) and Microsoft's Azure, it is not an abstraction of the MapReduce algorithm itself, which is the purpose of the tool presented in this paper. The three tools described above each have commonalities with MapRedoop, but the primary difference between each of the described tools and the approach presented here is the tools described above have been created for a very specific domain. The goal of MapRedoop is to provide an easier method to write MapReduce algorithms in a domain-independent manner.

[5]http://www.nofluffjuststuff.com/conference/reston/2011/04/session?id=20942
[6]http://mrflip.github.com/wukong/
[7]http://www.nvidia.com/object/cuda_home_new.html

Three additional tools presented by [Pike et al. 2005], McCullough, and Kromer are very similar to one another as they provide a DSL to write MapReduce algorithms more easily. These tools provide high-level abstractions to simplify the map and reduce functions; however, because of the simplification, they reduce the programmer's power. In contrast, MapRedoop simplifies the process of creating a MapRedoop program leaving the map and reduce function implementation to the programmer; thereby, maintaining a flexible and powerful environment.

4. A STUDY OF MAPREDUCE PROGRAMS

While implementing a MapReduce solution for a given problem, the programmer has to setup the data to allow the MapReduce framework to process the data efficiently, but the output generated from the framework may need to be converted. Section 4.1 explains such common setup steps involved in data conversions. Section 4.2 reveals more details about the interactions of MapReduce programs within our framework.

4.1 Steps Before and After MapReduce

The number of stages before and after the MapReduce execution differs based on the problem to be solved. As in the case of Hadoop[8], if the MapReduce implementation requires reading and writing to and from sequential files, more stages are required. Sequential files are flat files containing data in the key/value format. Sequential files support splitting up data for parallel jobs, even if they are compressed, making them a sufficient point of contact for the Hadoop framework. The input text files are read as Java objects and are written to sequential files before the MapReduce operation and converted back to text files after (or any other structure as needed).

To get a better understanding of the process, a detailed explanation is given for some of the common types of MapReduce problems. We categorize the MapReduce programs into three classes:

(1) **Class 1:** Programs that take text files as the input and read tokens (e.g., WordCount, Bigram, InvertedIndex [Lin and Dyer 2010]);
(2) **Class 2:** Programs that implement a machine learning algorithm (e.g., Clustering algorithms, Classifier algorithms); and
(3) **Class 3:** Programs that create a data structure (graph) internally for computation, e.g., PageRanking [Lin and Dyer 2010], and Breadth First Search (BFS). Please note: although some clustering algorithms take text files as input, we include it in the third class because the text files must first be converted to sequential files before passing to the MapReduce framework.

Programs that cannot be included either in the first or second classes are included in the third class (e.g., matrix multiplication or matrix transposition). The examples to demonstrate the process involved in implementing a MapReduce problem were carefully chosen such that the problems represent a general way of implementing a class of MapReduce programs. We selected InvertedIndex from the tokenizer algorithms (Class 1), the Clustering algorithm using Reuters benchmark[9] from the machine learning class (Class 2), and page ranking Wikipedia articles from the graph algorithms (Class 3). The focus of the analysis is to identify the steps involved in creating a MapReduce program not including the Map and Reduce sections.

4.1.1 *Inverted Index.* An InvertedIndex program for text files creates a data structure that maps words in the file to their locations [Zobel and Moffat 2006]. Another implementation of the InvertedIndex algorithm involves a data structure, which has a field to store the document identifier and a counter for each word in this data structure that is emitted from the mapper. The

[8] http://hadoop.apache.org/
[9] http://www.daviddlewis.com/resources/testcollections/reuters21578/

reducer collects the data structures for each word and combines the data to give the final inverted index of that word. The input is a text file, a data structure that is required to implement the MapReduce block, and a converter that is required to read the data structures to the required output structure (see Figure I).

4.1.2 *PageRank.* PageRank[10] is used by the Google search engine to sort search results [Page et al. 1998]. The algorithm assigns a weight to the pages based on the incoming and outgoing links in the documents. Pages are mapped to nodes while links are mapped to edges to create a graph structure. The algorithm works on the graph and weights are calculated for graph edges. The implementation[11] we used for our analysis was from Cloud9[12], a MapReduce library implemented using Hadoop for both teaching and data intensive research projects.

The program we used for our analysis creates a PageRank for all of the articles in the current version of Wikipedia. Wikipedia allows downloading[13] the article contents to a zip file, which can later be extracted to an XML file (e.g., 'date-pages-articles.xml'). As shown in Figure 1, the XML file is converted to a sequential file of type PageRankNode using the class RepackWikipedia. The MapReduce framework processes the sequential files and emits the output sequential files of type PageRankNode. Using a converter, the output sequential files are read as Java objects of type PageRankNode to get the data into the necessary final output structure.

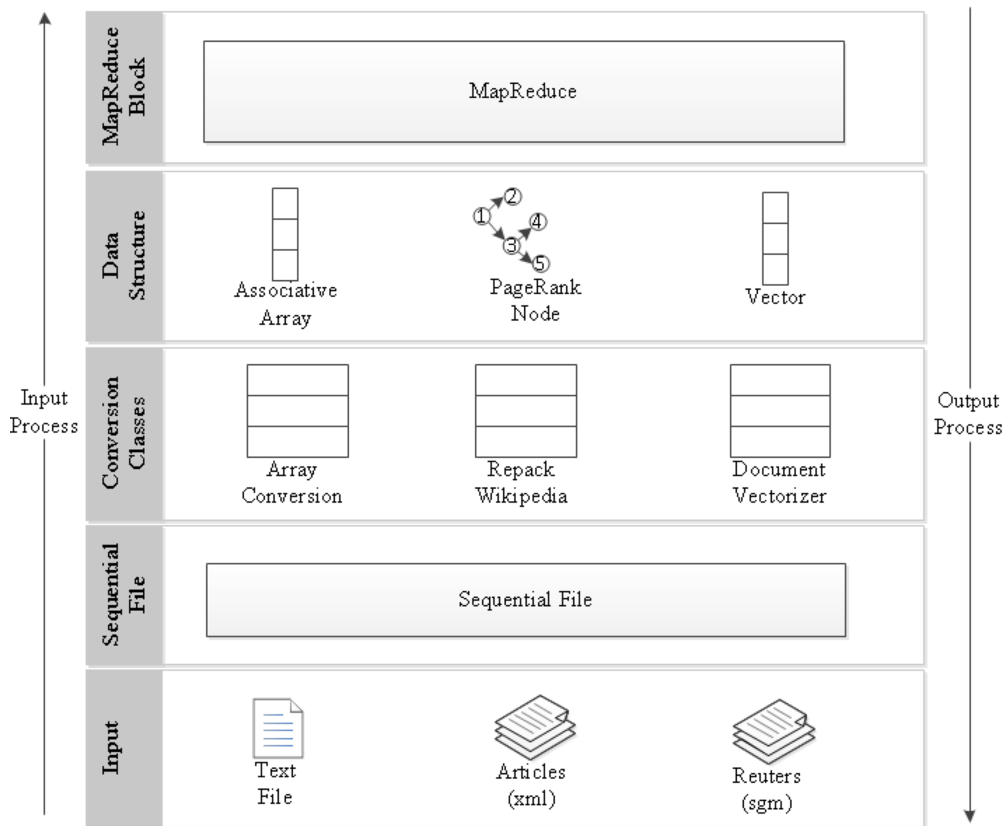

Figure. 1: Input/output process overview

[10]http://www.google.com/corporate/tech.html
[11]http://www.umiacs.umd.edu/ jimmylin/Cloud9/docs/content/pagerank.html
[12]http://www.umiacs.umd.edu/ jimmylin/cloud9/docs/
[13]http://dumps.wikimedia.org/enwiki/

4.1.3 *Clustering.* Clustering algorithms assign data into smaller groups based on a similarity factor. We used one of the simplest clustering algorithms, K-means. More details about K-means are presented in Section 6.2. We used the same implementation as Mahout. The K-means program is also commonly implemented using Hadoop. The program takes vectors as input and outputs a data structure of type Cluster. Using the data structure, the initial vectors can be clustered into specified groups, and the current center of each cluster can be determined.

The input used for the algorithm testing was Reuters-21578, one of the most widely used text collections in text categorization research. The text files are converted into sequential files of type String and are passed to a DocumentVectorizer, which parses the string to a Vector. The input vectors are then given to the MapReduce framework for clustering and output is generated as sequential files of type Cluster. Using a converter, `ClusterDumper`, the cluster sequential files are converted to text. The different stages of conversion involved in the K-means clustering of Reuters-21578 are shown in Figure 1.

4.2 Data Structure Analysis of MapReduce Programs

As mentioned earlier, writing a MapReduce solution includes specifying mappers and reducers. In some cases, adding a combiner and partitioner can make the solution more efficient. To define any of these, however, programmers are expected to specify the key type and value type of both the input and output. Hadoop has a small set of predefined key types and value types, which we found to be insufficient for providing solutions for real-world problems. For our analysis, all of the additional key or value types, which must be defined, are referred to as `Writable`. This is the interface Hadoop utilizes for defining new types. Some of the defined types might require a converter to read to and from Java Objects or text files. If they appear in the input of the mapper or output of the reducer, such types are referred to as 'Sequential types' (the types are generally written or read from sequential files). As an observation, if any of the input types of the mapper is a `Writable` object, the input structure for the MapReduce job will be a sequential file structure. The same is true for the output of the reducer.

Table I lists the various programs we used for our analysis and indicates whether the program has a mapper (M), reducer (R), combiner (C), and/or partitioner (P). The list of writable and sequential writable types for each program are also specified. All of the types shown as sequential writable are also writable; hence, in the table, it is shown as just sequential writable. The sequential writable types can be input (I), output (O), or both (I, O). The programs were collected primarily from Hadoop, Cloud9, and Mahout examples. If a program was collected from another source, the reference is provided within the table itself.

Class Type	Name	M	R	C	P	Writable types	Sequential Writable types
Class 1	WordCount	√	√	√			
	Bigram	√	√	√			
	InvertedIndex	√	√	√		AssociatedArray	
	Collocation matrix	√	√	√		Map	
Class 2	Collocation discovery	√	√	√		Gram	
	LDA model	√	√	√			Vector (I)
	Kmeans	√	√	√		ClusterObservation	Vector (I), Cluster (I,O)
	Dirichlet clustering	√	√			Vector (I)	Cluster (O)
	FuzzyKmeans	√	√	√		ClusterObservation	Vector (I), Cluster (I,O)
Class 3	HITS	√	√		√		HITSNode (O)
	PageRank	√	√	√			PageRankNode (I, O)
	BFS	√	√	√			BFSNode (I, O)
	Matrix multiplication	√	√	√			Vector (O)
	MonteCarlo[14]	√	√	√			GridJobResult (O)
	Image processing[15]	√	√				Image (O,I)

Table I: Data structure analysis of MapReduce programs

4.3 Summary of the analysis

The results of our analysis can be summarized in the context of three key concepts from software engineering engineering based on criteria for well-designed software, such as comprehensibility and reusability [Parnas and Clements 1986].

4.3.1 *Code comprehensibility.* Code comprehensibility plays a vital role both in software development and software maintenance [Deimel and Lionel 1985; Kernighan and Plauger 1982]. To understand the execution of a given MapReduce program implemented in Hadoop, which is often spread throughout many Java classes, the programmer must determine the Driver class. If proper naming conventions are not followed, the programmer has to find out the extended classes (`AbstractJob`) and implemented interfaces (`Writable`). This is the case for a programmer who is familiar with Hadoop APIs. Even a Java programmer familiar with MapReduce concepts has to read Hadoop documentation to get started with MapReduce programming.

The Driver class, which is the configuration file for Hadoop, is not sufficient in showing relevant information for the code reader in MapReduce programs. As mentioned in the motivation section, there exists a contract between Mapper, Reducer, and Combiner; the output of the mapper should have the same type as the input of reducer, or vice versa.

4.3.2 *Software reusability.* With the existing framework in Hadoop, MapReduce programs are written for a given input using a specified structure. This situation also occurs for libraries that are built over Hadoop. As an example, a BFS program from Cloud9 takes sequential files of type BFSNode as input. This type can be created from text files, which specify the graph in the following structure: "NodeId AdjacentNodeId1, AdjaceneteNodeId2." If a programmer desired to execute the BFS program where the input is specified in a slightly different structure (e.g., "NodeId: AdjacentNodeId, AdjacentNodeId"), he cannot directly use the BFS program from Cloud9.

4.3.3 *Generative programming.* Generative programming brings the benefit of automation to software development [Czarnecki and Eisenecker 2000]. Many programs require custom data structures (writable and sequential writable) to do the necessary computations. Implementing the `Writable` interface can be automated, as well as the corresponding converters, due to the type information about the output of the `Mapper` being redeclared in the input of the Reducer, and the input of Mapper being redeclared as parameters in the map method of the Mapper class.

Our analysis concludes that these issues occur due to the improper level of abstraction. Currently, MapReduce is implemented as an API, and these issues can be addressed if we can raise the level of abstraction. Type checking should be done at this new level of abstraction to allow the code to be more precise and readable. In the background, generative programming techniques can be used to run Hadoop while the programmer is presented with a small language targeted solely for MapReduce. Based on this analysis, a tool is presented that illustrates the potential for increasing the extensibility of input/output types in support of generative programming, which can assist in increasing code comprehensibility and software reusability.

5. MAPREDOOP

In this section, the MapRedoop framework is explained. MapRedoop has been used interchangeably to denote both the framework and the DSL. Section 5.1 explains the implementation of a MapReduce program using MapRedoop from a user's perspective and Section 5.2 explains the implementation of the MapRedoop framework.

5.1 Using MapRedoop in the Eclipse IDE

In this section, MapRedoop is explained from a user's perspective. The programmer writes the MapRedoop (DSL) for his/her current problem in a specialized editor (marked '7' in Figure 2), which supports syntax highlighting, code completion, validation and quick fixes, and advanced editor features such as bracket matching and outline view.

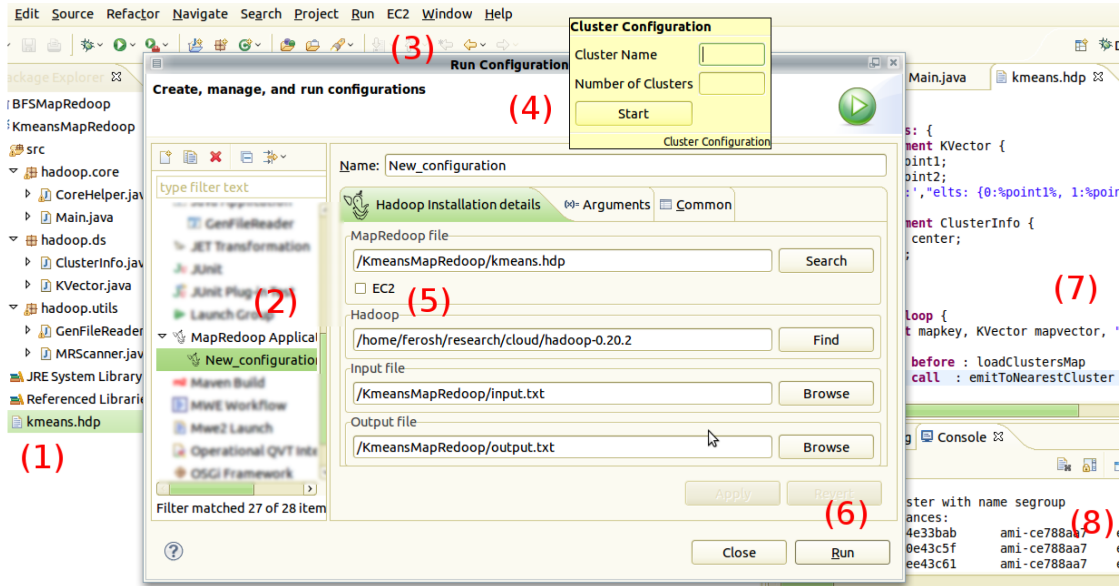

Figure. 2: Screenshot of MapRedoop in Eclipse IDE

Development of MapReduce programs in MapRedoop occurs in four stages:

(1) **Creating MapRedoop programs:** The programmer completes the MapRedoop program, which is represented by a file having extension ".hdp" (e.g., "kmeans.hdp" marked '1' in Figure 2). Within that program, the programmer specifies the required data structures and plug points in the framework.

(2) **Code generation:** The programmer generates code by right-clicking the ".hdp" file and using the "Generate code" option. This creates three Java packages: 1) hadoop.core, the main package for executing, setting up, and running the mapper and reducer, 2) hadoop.ds, the package for data structures in the program, and 3) hadoop.utils, the helper classes for converting text files to sequential files, and vice versa.

(3) **Implementing MapReduce methods:** The programmer implements the actual MapReduce algorithm. After the code generation, there are empty stub methods inside the "CoreHelper.java" classes. These empty methods give the full flexibility of the Java programming language for the programmer to implement the MapReduce logic.

(4) **Execution:** The programs written using MapRedoop can be executed in two modes:
   —*Hadoop standalone version:* Upon right-clicking the ".hdp" file and selecting the "Run as MapRedoop" option, the programmer is presented with a "Run Configuration" dialog as shown in Figure 2. If the option "EC2" (marked '5' in Figure 2) is unchecked, upon selecting "Run" (marked '6' in Figure 2), the program is executed as standalone. Because the run configuration (marked '2' in Figure 2) is implemented using the Eclipse run configuration framework[16], the same programs can be executed with different inputs or configurations, and these configurations can be saved.
   —*Hadoop cluster in EC2:* Before executing a program in EC2, a Hadoop cluster should be launched. This assumes that the required EC2 configurations have already been made in the Hadoop installation folder to start a Hadoop cluster in EC2. Upon selecting "EC2" (marked '3' in Figure 2), the programmer is presented with a light-weight pop-up (marked 4' in Figure 2) to specify the cluster name and number of slaves.

[16]http://www.eclipse.org/articles/Article-Launch-Framework/launch.html

All communication between the Hadoop server, whether it is standalone or EC2, are shown in the console (marked '8' in Figure 2). Video demonstrations of standalone executions and EC2 clusters can be viewed at our project web site (see footnote 2).

5.2 High-Level Design Diagram

A high-level design diagram of the MapRedoop framework is shown in Figure 3.

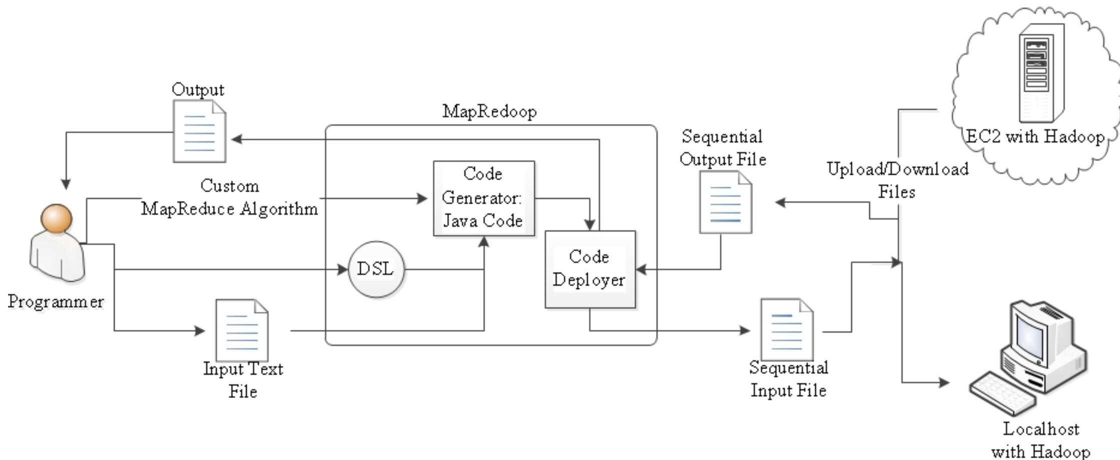

Figure. 3: MapRedoop Overview

The configuration and control of this framework is achieved through the MapRedooop DSL explained in Section 5.3. As shown in Figure 3, the MapRedoop tool has two components:

(1) **Code generator:** This component takes the MapRedoop DSL as input and generates code for three packages, as mentioned previously. These generated classes can be combined into three categories:
   (a) *Core classes:* These include the Mapper, Reducer, Combiner, Partitioner, and the Driver. Other than the Driver class, these classes are only generated if they are mentioned in the MapRedoop DSL. In addition to the driver class, a `CoreHelper` class is generated; this is where the programmer will implement the actual MapReduce algorithm.
   (b) *Data structure classes:* Data structure classes are new types defined in the MapRedoop DSL. There can be two types of data structure classes: 1) Data structure types that occur as the key/value of any of the mappers or reducers, and 2) Data structure types that are only used inside the program. In Hadoop, data structure types that occur as the key/value pairs require Hadoop to implement an interface called `Writable`.
   (c) *File conversion classes:* Programmers can specify the template in which input files should be read. The MapRedoop framework generates classes such that the data can be read while converting the text file to a sequential file and also while converting the sequential file back to a text file. The classes are executed before and after the execution of the MapReduce programs to make the automatic conversion of data possible.

(2) **Code deployer:** The deployment is done in four stages:
   (a) *Target environment:* Based on the user's choice, the tool deploys the code either in Hadoop standalone version or in the EC2 Hadoop cluster.
   (b) *Format conversion I:* Text files or the input information has to be converted to sequential files, and in the case of the user selecting the EC2 cluster, the sequential files must be uploaded to the server.
   (c) *Communication:* Collect the results from the server in case of a cluster deployment.
   (d) *Format conversion II:* Convert the results back to the original input structure.

The key/value types of the program determine the steps. If the key-value types are not composite, as in the case of the simple 'WordCount' program, the deployer just uploads the input files to the server, executes the results and collects the output to the requested folder. Class 1 programs generally skip the conversion steps because the input for the programs are text files.

5.3 MapRedoop DSL

A subset of the grammar for the MapRedoop DSL is shown in Figure 4 (only the important parts of the grammar are shown for clarity).

```
1. MapRedoop: Declaration { (Content)*}
2. Declaration: program ID (extend ID)?
3. Content: ListofEntities | MRBlock
  a. ListofEntities: metaelements: { (Entity)* }
    i. Entity: metaelement ID (extend [Entity])? { (Feature)+ }
    ii. Feature: TypeDeclaration; | ReadWrite;
   iii. ReadWrite: read (STRING, STRING) | write (STRING, STRING)
  b. MRBlock: mapreduce: (loop)? { Mapper Reducer }
    i. Mapper : map(Argument, Argument, STRING, STRING ) Block
    ii. Reducer : reduce(STRING,STRING, STRING , STRING') Block;
    iii. Block: [(JavaMethodCall)* ]
    iv. JavaMethodCall: TimeOfCall : ID
    v. TimeOfCall: after| call | before
```

Figure 4. MapRedoop DSL grammar

The MapRedoop program starts with a declaration statement (line 2). An optional 'extend' is included to reuse some of the features already declared within a DSL. All of the data structures declared in the parent DSL will be available inside the extending DSL (child DSL). The DSL has two sections: 1) Meta-elements (line 3a), and 2) MapReduce (line 3b).

(1) **Meta-elements:** This block is for generating Java code for the data structures to be used later by file structure conversion classes or the MapReduce framework itself. For every `metaelement` defined in the `metaelements` section, the code generator checks whether that data structure appears as a type in the keys or values of the mapper or reducer. If it does not appear, it is generated as a regular Java class with the fields declared in the `metaelement`, adding setter and getter methods. If the types appear in any of the key/value pairs of mapper or reduce, the class is generated as a `Writable` Java class.

(2) **Writable classes:** In Hadoop, the classes that are used as a key or value for the mappers or reducers must implement the `Writable` interface. In order to implement this interface, the programmer has to implement two methods, `read` and `write`. An example implementation[17] of these methods is shown in Figure 5. The code block is taken from a class having three fields (`x, y`, and `z`), each of type float. As demonstrated in Figure 5, it is clear that if we know the field types, the code can be generated. Aggregate relationships, such as a Java List, are implemented using an array of elements and each element's type. In this situation, our implementation adds an additional variable called `fieldname+size` of type `int` to the original list of fields.
Each meta-element can have two types of features: 1) A `TypeDeclaration` (`metaelement` or native type, such as int) and 2) `ReadWrite`. We first define the fields in a data structure and later link them to the input structure in a text file. A line that is coded as:

```
read (‘‘ ’’, %nodeId%{%distanceFromSource%} : %adjacentlist%" );
```

[17] http://developer.yahoo.com/hadoop/tutorial/module5.html

from a meta-element Node having three fields: `nodeId, distanceFromSource, adjacentlist,` declares that the input structure for reading this object originates from a text file. Given this information, during runtime, the parsers can create Java objects with the values read from a text file. As an example, if a file has the following two lines: "2{3} : 3" and "3{2} : 4 5 6", two Java objects are created:

(a) The first object created is `nodeId` 2 consisting of `distanceFromSource` 3 and a list `adjacentlist` having the value of 3.
(b) The second object created is `nodeId` 3 consisting of `distanceFromSource` 2 and a list `adjacentlist` having 4 5 6.

The second parameter of the read method defines the structure and the first parameter is an optional way of specifying a delimiter while parsing a list of values. For objects of type `adjacentlist`, a space is used as the delimiter.

(3) **MapReduce blocks:** The MapReduce blocks were designed to avoid the repeated declaration of types required in the Hadoop implementation. Hence, there is no input declaration for reducer and input/output declaration for combiner, because these would be the same as the output of the mapper. Therefore, reducer has no input declaration, only output declaration. The `map` function takes two parameters (type followed by variable name) and two arguments (both representing a type). The first two parameters declare two variables, and those variables can be used inside the mapper function. The last two parameters are arguments declaring the output type of the mapper. Those types are only used inside the reducer; hence, those variable names are defined as the first two arguments of the `reducer`. The last two parameters of the reducer define the output types of the MapReduce program.

```
1.    public void write(DataOutput out) throws IOException {
2.      out.writeFloat(x);
3.      out.writeFloat(y);
4.      out.writeFloat(z);
5.    }

6.    public void readFields(DataInput in) throws IOException {
7.      x=in.readFloat();
8.      y=in.readFloat();
9.      z=in.readFloat();
10.   }
```

Figure. 5: Sample `read` and `write` implementation of `Writable`

In addition to the above, the MapReduce block allows plug-in Java calls to implement the actual MapReduce program. For `reducer` and `mapper`, these Java calls can be made either during the process function, or before or after the core function call. Plug-in method calls are generated to support iterations of MapReduce calls by setting the flag loop. Examples of two programs written using MapRedoop are introduced in Section 6.

6. CASE STUDIES

Algorithm	K-Means			BFS		
Tool	MapRedoop	Mahout	% Reduction	MapRedoop	Cloud9	%Reduction
Lines of Code	99 + 23	493	75%	94 + 20	331	66%

Table II: Lines of Code comparison of Hadoop libraries with MapRedoop

In this section we describe two algorithms implemented using MapRedoop and compare the solutions to other Hadoop libraries. Implementing the K-means clustering algorithm while implementing the Breadth First Search (BFS) algorithm requires Class 3 types (please see Table I). Code-level comparison of the programs is shown in Table II. Lines of code for MapRedoop represent the actual implementation added to the MapRedoop DSL code. The lines of code for the libraries include the code specifically written for implementing the program. The pre-defined data structures (e.g., `Vector` in Mahout) are not considered when counting the total lines of code. As described in the table, MapRedoop requires 75% less lines of code to implement a BFS algorithm as compared to Cloud9, and 66% less lines of code to implement a K-means algorithm as compared to Mahout. A detailed performance comparison of the two case studies along with their MapRedoop solution is given in the following sub-sections. For the performance analysis, we executed the two versions (MapRedoop and Hadoop Library) of the program in a standalone Hadoop installation and also in an EC2 Hadoop cluster. The clusters were implemented with one, two, four, and eight slaves for a given size of data. Every execution in the cluster, as well as the standalone versions, was executed three times and the reading taken was the mean of the three executions.

BFS program

```
1  program BFS {
2      metaelements: {
3          metaelement Node {
4           long nodeId;
5           long distanceFromSource  ;
6           int nodeType ;
7           int* adjacentlist;
8           read(':',"%nodeId%{%distanceFromSource%} : %adjacentlist%" );
9          }
10     }
11     mapreduce: loop {
12         map(text mapkey, Node mapnode, "long" , "Node" ) [
13                 call  : emitStructure
14                 after : emitDistance
15             ]
16         reduce("redkey", "nodes", "long", "Node")[
17                 call: minmizeDistance
18             ]
19     }
20 }
```

Figure. 6: MapRedoop DSL for BFS in Eclipse IDE

6.1 Implementing the Breadth First Search algorithm in MapRedoop

Breadth First Search (BFS) is a common algorithm to find the distance from the source node to all the reachable nodes. The algorithm begins by finding all of the neighbors for the first node, and for each of the first node's neighbors, the algorithm finds those neighbors. This cycle continues until the algorithm reaches the goal node. Applications of BFS include finding the shortest path and spanning forests.

Implementation of BFS in MapReduce involves finding the distance from the current node to the source node. The mapper is responsible for storing the computed distance to the next node. This node is then passed on to the reducer and emitted. The reducer collects all of the nodes from the mapper, and for each node, the reducer selects the node storing the smallest distance. For every node, the least distant node is selected. Each MapReduce iteration is a hop in the graph. The mapper should emit the structure along with the distance so that the adjacent nodes

can be calculated for the next iteration. A MapRedoop DSL for describing these properties is shown in Figure 4.

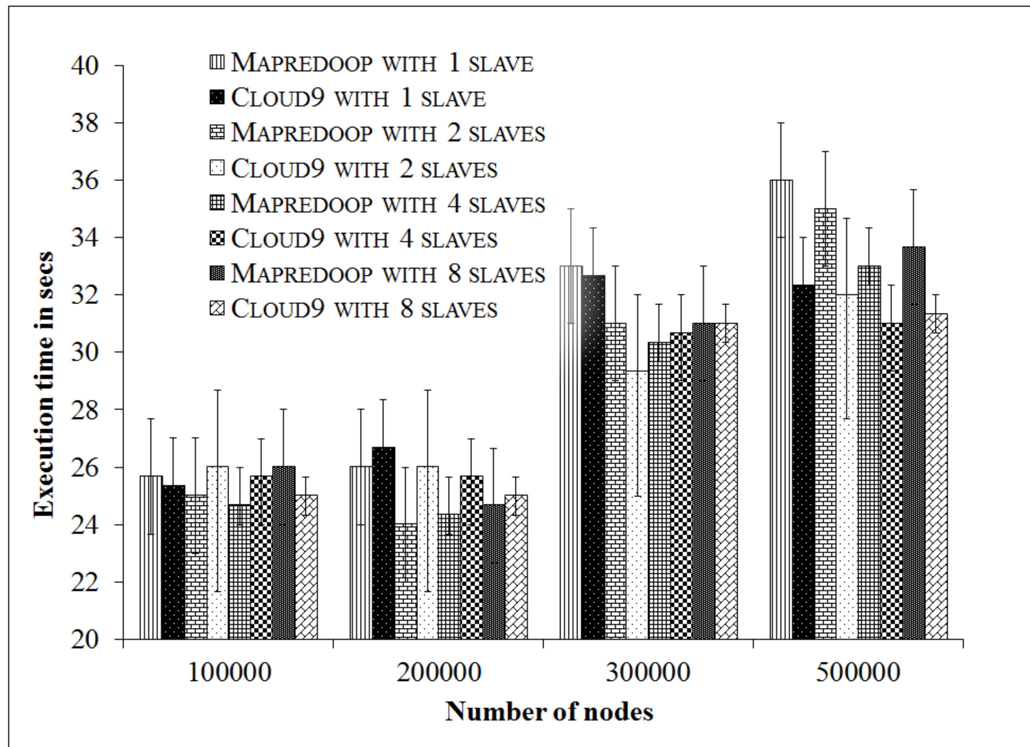

Figure. 7: Execution time of BFS programs in EC2 Hadoop cluster

6.1.1 *MapRedoop DSL.* The MapRedoop solution for the BFS algorithm is shown in Figure 6. In the meta-elements block, a data structure or meta-element is declared called `Node` with fields: `nodeId, distanceFromSource, nodeType,` and `adjacentList`. A Java class will be generated from this structure with the name and corresponding fields of each type, as mentioned in the DSL. While reading the structure "2{3}: 3, 4" from a file, the read method creates a Node object with `nodeId` of 2, `distanceFromSource` of 3, and `adjacent nodes` 3 and 4. In the MapReduce block, the `map` function accepts `mapkey` of type `Text` as the input key, and the input value `mapnode` of type `Node`. The `mapper` function sets up an output key of type `long`, and creates an output value of type `Node`.

As shown in the BFS example in Figure 6, only the type is defined in the `map` function. The name of the variables are defined in the `reduce` function, because reducer uses the variables while the map function defines the type of the variables. The `reduce` function also defines the type of the output key and value. On code generation, MapRedoop creates `Mapper` and `Reducer` classes, along with the `Driver` class containing all the necessary key value declarations. A class called `CoreHelper` is created with methods `emitStructure, emitDistance,` and `minimizeDistance`. The code is configured such that the map method in the `Mapper` class calls the `emitStructure` method, while the `cleanup` method in the `Mapper` class calls the `emitDistance`. Finally, the `reduce` method in the `Reducer` class calls the `minimizeDistance` method.

6.1.2 *Performance Analysis.* The data for the standalone version was classified into four categories: 1) Micro: a graph having 2,000 nodes and a variables number of edges (0-5), 2) Small: graph having 10,000 nodes, each having 5 edges, 3) Medium: a graph having 50,000 nodes, each

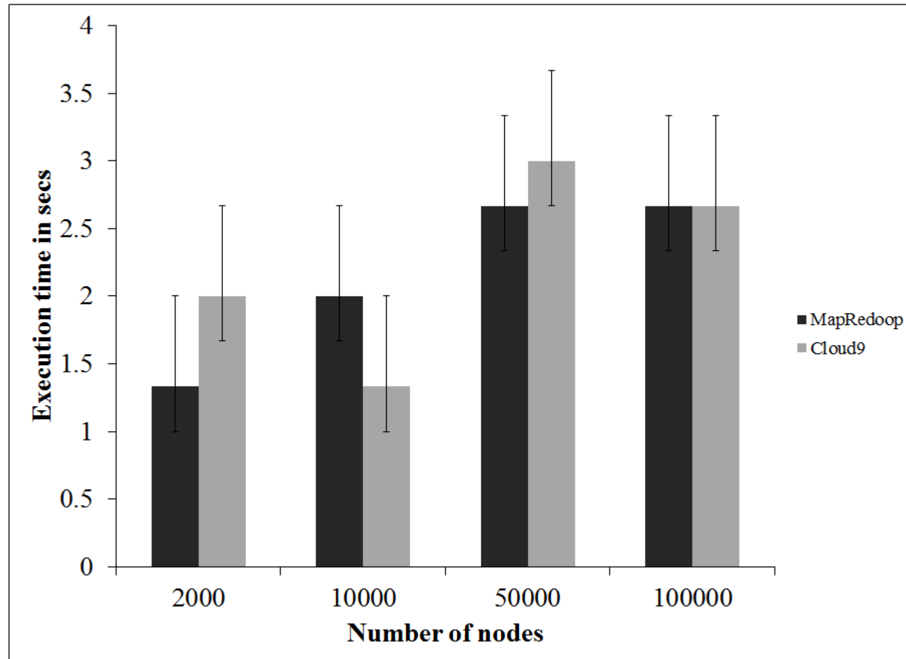

Figure. 8: Execution time of BFS programs in Hadoop standalone mode

having 5 edges, 4) Large: a graph having 100,000 nodes, each having 5 edges. For the cluster micro, small, large, medium, large had 100,000, 200,000, 300,000, and 500,000 nodes respectively. For performance comparison, we selected a program written by expert Hadoop programmers from Cloud9, which was implemented using the same algorithm. From the figures shown in Figure 7 and Figure 8, both the MapRedoop and the expert-created program gave comparable performance in both the standalone and cluster implementations. As the graphs illustrate, there are some scenarios where MapRedoop performed more poorly than Cloud9. The bars represent the average of the trials ran, and the error bars represent the max and min values of the trials. In some cases there is a large delta between the max and min illustrating the inconsistency of the Hadoop File System, which is part of the reason for the vast difference in performance from scenario to scenario. Additionally, due to the added flexibility for input/output types provided by MapRedoop, there is a degradation in runtime.

6.2 Implementing the K-means algorithm in MapRedoop

K-means is possibly one of the most commonly used clustering algorithms according to [Kanungo et al. 2002]. The K-means clustering algorithm groups a cluster into 'k' small clusters based on a similarity factor. The similarity factor we used in this implementation is the distance. The algorithm starts with randomly selected 'k' vectors (in our case, user specified vectors). For every input vector, the algorithm calculates the distance from the initial 'k' vectors to the current vector, and the closest 'k' vector is grouped with the current vector. In the MapReduce implementation of K-means, every vector in the mapper part is emitted to the nearest cluster and the reducer part collects the vectors to a given cluster.

6.2.1 *MapRedoop DSL.* The DSL for implementing K-means using MapRedoop is shown in Figure 9. In addition to the features explained in the previous context of BFS, a K-means program makes use of other features of MapRedoop. The `ClusterInfo` meta-element is a special type that does not occur in any of the key/value types of the mapper or reducer. Hence, `ClusterInfo` is generated as an ordinary Java class (not an implementation of `Writable`) and `KVector` is

Kmeans Program

```
program Kmeans{
    metaelements: {
        metaelement KVector {
        float point1;
        float point2;
         read(':',"elts: {0:%point1%, 1:%point2%}" );
        }
        metaelement ClusterInfo {
        KVector center;
        long id;
        }
    }
    mapreduce: loop {
        map(text mapkey, KVector mapvector, "long" , "KVector" ) [
                before : loadClustersMap
                call   : emitToNearestCluster
            ]
        reduce("redkey", "clustervalues", "long", "KVector")[
                before : loadClustersRed
                call: calculateNewCenter
            ]
    }
}
```

Figure. 9: MapRedoop DSL for K-means in Eclipse IDE

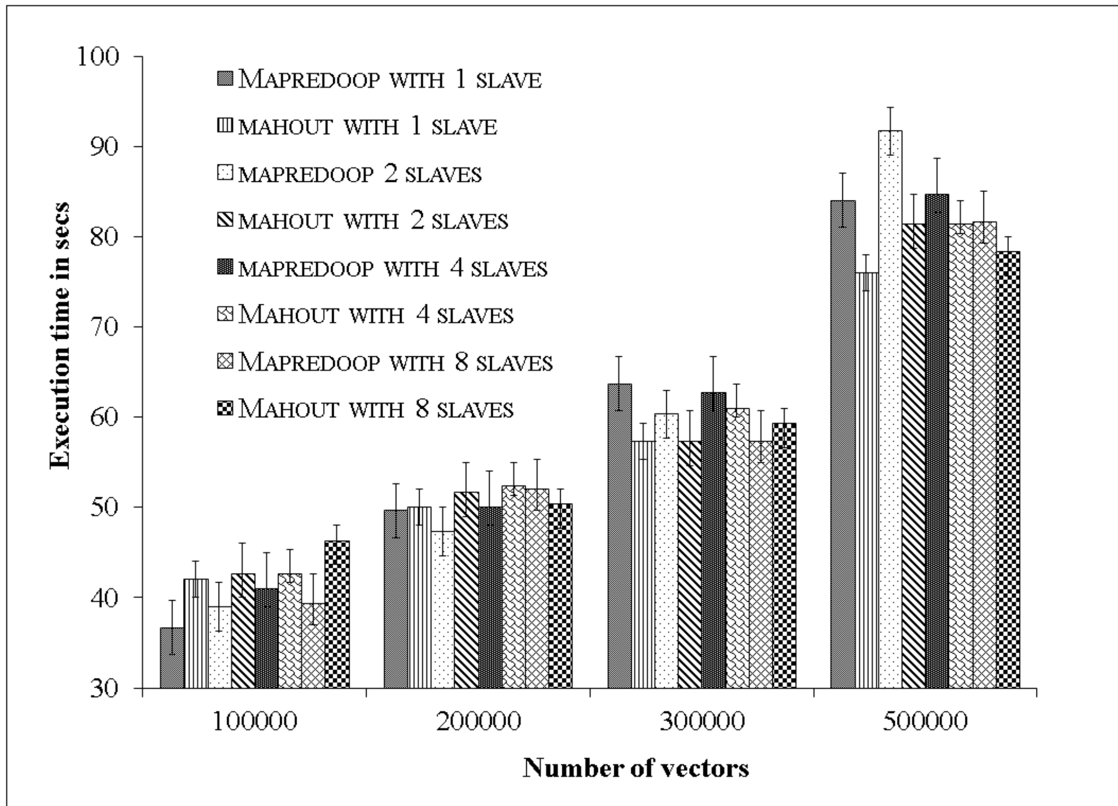

Figure. 10: Execution time of K-means programs in EC2 Hadoop cluster

generated as a `Writable` Java class. In this case, the `before` keyword is used both in the `map` and `reduce` blocks. Hence, two additional methods are created in the `CoreHelper` class, which is called from `setup` functions of `Mapper` and `Reducer` classes. In the implementation of K-means,

there is an additional input other than the input vectors that represent the current clusters. Before invoking the map and reduce operations, the data from the current clusters has to be loaded using `loadClustersMap`.

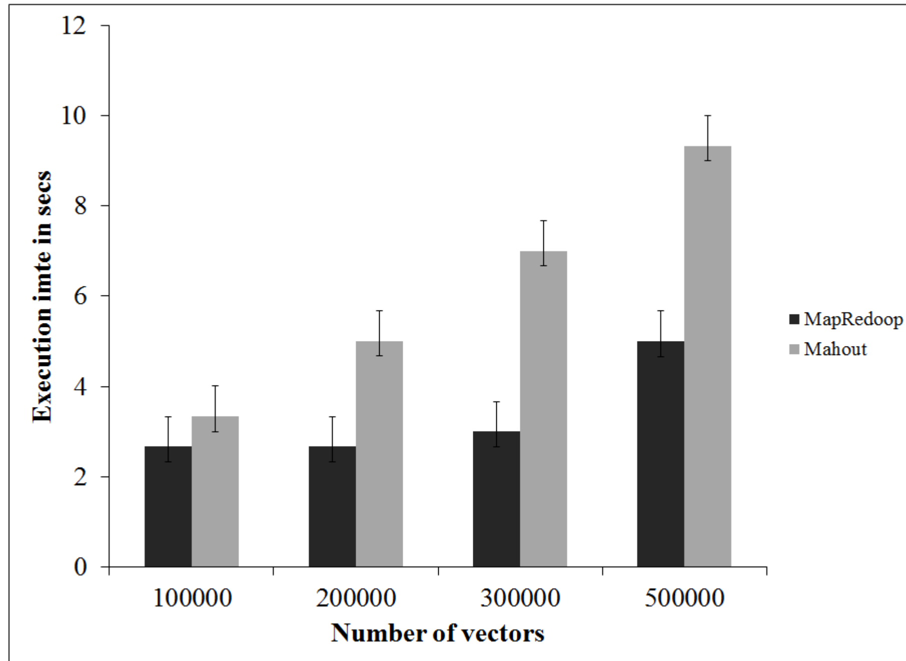

Figure. 11: Execution time of K-means programs in Hadoop standalone mode

6.2.2 *Performance analysis.* The K-means implementation from the Mahout project was used for a performance comparison. Both programs produce the same output. As input, N-points were used to group the output into three clusters based on their Manhattan distance, which is the distance measured along axes at right angles. The execution plots of these algorithms in EC2 and standalone are presented in Figures 10 and 11. Four different values were used for N : 1) 100,000, 2) 200,000, 3) 300,000, and 4) 500,000. The MapRedoop version dominated in the standalone version and had comparable results in the cluster implementation. This can be attributed to the Vector data structure in Mahout. Because the K-means solution in Mahout is a case of a generic clustering solution, there are many fields in the data structure that are not relevant to the K-means problem. This can result in more writing and reading during file operations. In the case of MapRedoop, solutions are written for a problem, and hence the programmer needs to define only the fields relevant to the problem. As mentioned in the previous analysis, the inconsistency in the Hadoop File System and the added input/output flexibility contribute to MapRedoop's degraded performance.

7. CONCLUSION AND FUTURE WORK

After writing several MapReduce programs in Hadoop, we recognized three specific areas of inefficiency resulting from accidental complexities: input structure inflexibility, level of abstraction, and ease of testing. Our goal was to create a tool that would provide an easier and faster means for programmers to write MapReduce algorithms without affecting performance. Our solution, MapRedoop, is a framework implemented in Hadoop that combines a DSL and IDE that removes the encountered accidental complexities. To evaluate our tool, we implemented two commonly described algorithms (BFS and K-means) and compared the execution of MapRedoop to existing

methods (Cloud9 and Mahout). With MapRedoop, the programmer only needs to code the DSL and MapReduce algorithms, whereas Cloud9 and Mahout focused on input/output conversions. The analysis presented in Section 6 illustrate that MapRedoop performs comparably to the existing, common methodologies, and in some cases, MapRedoop proved to have better performance due to the programmer being able to focus solely on the problem.

To expand upon this project in the future, we plan to implement support for additional MapReduce implementations, such as Phoenix [Ranger et al. 2007] or Mars [He et al. 2008]. Additionally, we want to increase the flexibility of MapRedoop by bridging it with the Hadoop Eclipse plug-in to allow MapRedoop programs to be deployed in any cluster that supports Hadoop. Finally, the primary benefit of MapRedoop is to the programmer. We plan to perform an empirical study to demonstrate the time saved using MapRedoop.

REFERENCES

Allen, E., Chase, D., Hallett, J., Luchangco, V., Maessen, J.-W., Ryu, S., Jr., G. L. S., and Tobin-Hochstadt, S. 2007. The Fortress Language Specification. Tech. rep., Sun Microsystems, Inc.

Chafi, H., Sujeeth, A. K., Brown, K. J., Lee, H., Atreya, A. R., and Olukotun, K. 2011. A domain-specific approach to heterogeneous parallelism. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*. San Antonio, TX, 35–46.

Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., and Sarkar, V. 2005. X10: an object-oriented approach to non-uniform cluster computing. *ACM SIGPLAN Notices 40,* 10 (October), 519–538.

Czarnecki, K. and Eisenecker, U. 2000. *Generative programming: methods, tools, and applications.* ACM Press/Addison-Wesley Publishing Co. New York, NY.

Dean, J. and Ghemawat, S. 2004. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the Symposium on on Operating Systems Design & Implementation.* USENIX Association, San Francisco, CA, 137–150.

Dean, J. and Ghemawat, S. 2008. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM 51,* 1 (January), 107–113.

Deimel, J. and Lionel, E. 1985. The uses of program reading. *ACM SIGCSE Bulletin 17,* 2 (June), 5–14.

Diaconescu, R. and Zima, H. 2007. An approach to data distributions in chapel. *Internation Journal of High Performance Computing Applications 21,* 3 (August), 313–335.

Fritz, N., Lucas, P., and Slusallek, P. 2004. CGiS, a new language for data-parallel gpu programming. In *Proceedings of the 9th International Workshop Vision, Modeling, and Visualization.* Stanford, CA, 241–248.

Ghemawat, S., Gobioff, H., and Leung, S.-T. 2003. The google file system. In *Proceedings of the Symposium on Operating systems principles.* ACM, Bolton Landing, NY, 29–43.

He, B., Fang, W., Luo, Q., Govindaraju, N. K., and Wang, T. 2008. Mars: A MapReduce framework on graphics processors. In *Proceedings of the Innternational Conference on Parallel Architectures and Compilation Techniques.* ACM, Toronto, Ontario, Canada, 260–269.

Jacob, F., Arora, R., Bangalore, P., Mernik, M., and Gray, J. 2009. Raising the level of abstraction of gpu-programming. In *Proceedings of the 16th International Conference on Parallel and Distributed Processing Techniques and Applications.* Las Vegas, Nevada, 339–345.

Jacob, F., Whittaker, D., Thapaliya, S., Bangalore, P., Mernik, M., and Gray, J. 2010. Cudacl : A tool for cuda and opencl programmers. In *Proceedings of the International Conference of High Performance Computing.* Goa, India, 1–11.

Kanungo, T., Mount, D. M., Netanyahu, N. S., Piatko, C. D., Silverman, R., and Wu, A. Y. 2002. An efficient k-means clustering algorithm: Analysis and implementation. *IEEE Transactions on Pattern Analysis Machine Intelligence 24,* 7 (July), 881–892.

Kernighan, B. W. and Plauger, P. J. 1982. *The Elements of Programming Style*, 2nd ed. McGraw-Hill, Inc., New York, NY.

Lin, J. and Dyer, C. 2010. *Data-Intensive Text Processing with MapReduce.* Synthesis Lectures on Human Language Technologies. Morgan & Claypool Publishers.

Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., and Hellerstein, J. M. 2010. Graphlab: A new framework for parallel machine learning. *Clinical Orthopaedics and Related Research abs/1006.4990.*

Manjunatha, A., Anderson, P., Ranabahu, A., and Sheth, A. 2011. Identifying and implementing the underlying operators for nuclear magnetic resonance based metabolomics data analysis. In *Proceedings of the International Conference on Bioinformatics and Computational Biology.* ACM, New Orleans, LA.

Mernik, M., Heering, J., and Sloane, A. M. 2005. When and how to develop domain-specific languages. *ACM Computing Surveys 37,* 4 (December), 316–344.

PAGE, L., BRIN, S., MOTWANI, R., AND WINOGRAD, T. 1998. The PageRank citation ranking: Bringing order to the web. Tech. rep., Stanford Digital Library Technologies Project.

PARNAS, D. L. AND CLEMENTS, P. C. 1986. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering 12,* 2 (February), 251–257.

PIKE, R., DORWARD, S., GRIESEMER, R., AND QUINLAN, S. 2005. Interpreting the data: Parallel analysis with sawzall. *Scientific Programming 13,* 4 (October), 277–298.

RAJA, A. AND LAKSHMANAN, D. 2010. Article: Domain specific languages. *International Journal of Computer Applications 1,* 21 (February), 99–105.

RANABAHU, A., SHETH, A., MANJUNATHA, A., AND THIRUNARAYAN, K. 2010. Towards cloud mobile hybrid application generation using semantically enriched domain specific languages. In *International Workshop on Mobile Computing and Clouds*. ACM, Santa Clara, CA.

RANGER, C., RAGHURAMAN, R., PENMETSA, A., BRADSKI, G., AND KOZYRAKIS, C. 2007. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proceedings of the International Symposium on High Performance Computer Architecture*. IEEE Computer Society, Phoenix, AZ, 13–24.

SUGIKI, A., KATO, K., ISHII, Y., TANIGUCHI, H., AND HIROOKA, N. 2010. Kumoi: A high-level scripting environment for collective virtual machines. In *International Conference on Parallel and Distributed Systems*. Vol. 0. IEEE Computer Society, Shanghai, China, 322–329.

SUJEETH, A. K., LEE, H., BROWN, K. J., ROMPF, T., CHAFI, H., WU, M., ATREYA, A. R., ODERSKY, M., AND OLUKOTUN, K. 2011. OptiML: An implicitly parallel domain-specific language for machine learning. In *Proceedings of the International Conference on Machine Learning*. Haifa, Israel.

WU, H. AND GRAY, J. 2005. Testing domain-specific languages in eclipse. In *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM, San Diego, CA, 173–174.

ZOBEL, J. AND MOFFAT, A. 2006. Inverted files for text search engines. *ACM Computing Surveys. 38,* 2 (July).

Ferosh Jacob is a Ph.D. student at the University of Alabama in the Computer Science department. His research is currently focused on applying software engineering techniques to computation intensive problems. In 2007, he completed his Master's degree in Computer Science at Clarkson University. He received his B.Tech. degree in Electronics and Communication at the National Institute of Technology, Calicut India. He can be reached at fjacob@crimson.ua.edu.

Amber Wagner is a Ph.D. student in the Department of Computer Science at the University of Alabama. She completed her Master's degree from Kennesaw State University in 2008 after teaching High School Computer Science in Birmingham, Alabama. Her research is focused on integrating voice driven computer interaction via modeling. Additionally, she is focused on community outreach to improve secondary Computer Science education in the state of Alabama. She can be reached at ankrug@bama.ua.edu.

Prateek Bahri is a Ph.D. student in the Department of Computer Science at the University of Alabama. His current research involves the development and improvement of testing techniques for smartphone devices. He holds a B.Tech degree in Computer Science from Chitkara University, India (2002-2006). He has experience in a software company as a web-developer(2006-2010). He can be contacted at pbahri1@crimson.ua.edu.

Susan V. Vrbsky is an Associate Professor in the Department of Computer Science at The University of Alabama. She received her Ph.D. in Computer Science from The University of Illinois, Urbana-Champaign. Her research interests include data grids, data management in clouds, green computing, real-time database systems and database security. She is the faculty advisor to the Cloud and Cluster Computing Laboratory at UA. She can be reached at vrbsky@cs.ua.edu.

Jeff Gray is an Associate Professor in the Department of Computer Science at the University of Alabama (UA). He co-directs the newly formed Software Engineering Group at UA and currently serves as the chair of the Alabama IEEE Computer Society. He received his Ph.D. from Vanderbilt University and a M.S./B.S. in Computer Science from West Virginia University. Jeff's research interests are in model-driven engineering, software evolution, and generative approaches that support automation. Jeff is a Senior member of IEEE and a member of ACM. He can be reached at gray@cs.ua.edu.