Mrs: MapReduce for Scientific Computing in Python

Andrew McNabb, Jeffrey Lund, and Kevin Seppi
Computer Science Department
Brigham Young University
Provo, UT 84602
Email: a@cs.byu.edu, jefflund@byu.edu, k@byu.edu

***Abstract*—The MapReduce parallel programming model is designed for large-scale data processing, but its benefits, such as fault tolerance and automatic message routing, are also helpful for computationally-intensive algorithms. However, popular MapReduce frameworks such as Hadoop are slow for many scientific applications and are inconvenient on supercomputers and clusters which are common in research institutions.**

Mrs is a Python-based MapReduce framework that is well suited for scientific computing. We present comparisons of programs and run scripts to argue that Mrs is more convenient than Hadoop, the most popular MapReduce implementation. We also demonstrate that Mrs outperforms Hadoop for several types of problems that are relevant to scientific computing. In particular, Mrs demonstrates per-iteration overhead of about 0.3 seconds for Particle Swarm Optimization, while Hadoop takes at least 30 seconds for each MapReduce operation, a difference of two orders of magnitude.

I. Introduction

MapReduce [1] has quickly become a popular paradigm for large scale data intensive analysis and has also been applied to computationally intensive programs. It has been used for iterative algorithms such as k-means [2], logistic regression [3], backpropagation [3], independent component analysis [3], expectation maximization [3], support vector machines [3], genetic algorithms [4], and particle swarm optimization (PSO) [5]. The popularity of MapReduce may be attributed to its simplicity and availability.

Unfortunately, most current MapReduce frameworks exhibit poor performance in scientific applications [2]–[5] and are ill suited to the computational environments that are most important for scientific computing. Many universities and research institutions have supercomputers, but these are not tied to any particular parallel processing technology. Most popular MapReduce frameworks are designed for large-scale data processing in datacenters and require a dedicated cluster and extensive configuration. Such frameworks use technologies that make MapReduce unnecessarily complex to program and difficult to run on supercomputers and clusters that are common in research institutions. Furthermore, performance is not optimized for computationally intensive applications, particularly iterative algorithms.

Mrs is a lightweight Python-based MapReduce implementation. It is designed to make MapReduce programs easy to write, easy to run, and fast. Python helps make these design goals possible. Mrs programs are easy to write because of the convenience and readability of Python. The Mrs API is also designed to avoid the need for unnecessary boilerplate. Mrs programs are easy to run because it relies only on the Python standard library and works with any job scheduler or filesystem. Mrs programs are fast because Mrs is the product of multiple significant rewrites to improve efficiency and reduce overhead, and Python makes such restructuring manageable. Furthermore, Python provides powerful approaches for accelerating programs without sacrificing simplicity, such as running in PyPy or integrating with custom C modules. All of these strengths contribute to making Mrs an effective platform for scientific computing.

Mrs offers both subjective improvements and performance improvements over Hadoop, the most popular MapReduce framework. We evaluate the ease of programming and readability by comparing a Mrs program written in Python with a functionally equivalent Hadoop program written in Java. We show the simplicity in running a Mrs job on a supercomputer with a PBS job scheduler relative to a Hadoop job in the same environment. We also demonstrate the performance advantages of Mrs over Hadoop using three applications: WordCount in Project Gutenberg, a collection of 31,173 documents; a computationally intensive estimator of π ranging from 1 to 10^9 samples; and Particle Swarm Optimization, an empirical function optimization algorithm. In all cases, Hadoop exhibits significant overhead. Despite the inherent performance advantage of Java over Python, the Mrs program maintains a significant performance advantage when task times are less than around 32 seconds, which is extended to around 40 seconds when using a C module in the innermost loop and using the PyPy interpreter. This performance advantage is particularly significant in the context of iterative algorithms, where overhead is incurred each iteration.

Section II reviews the MapReduce programming model and other MapReduce implementations. Section III discusses the context of scientific computing and the specific needs that it requires of a MapReduce implementation. Section IV describes the programming model and the design of Mrs, including the advantages and challenges of using Python to implement a MapReduce system. Section V presents results showing the benefits of Mrs over Hadoop.

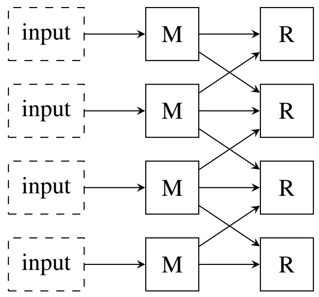


Fig. 1: Task dependencies of the map (M) and reduce (R) operations in a MapReduce program.

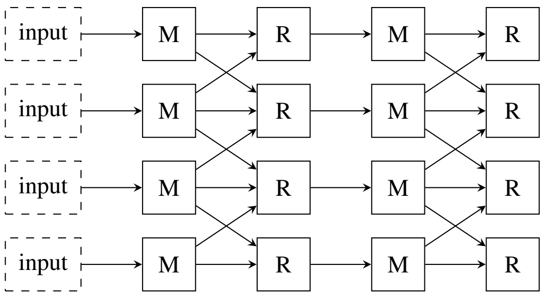


Fig. 2: Task dependencies of the map (M) and reduce (R) operations in an iterative MapReduce program.

II. Background and Related Work

MapReduce is a functional programming model that is well suited to parallel computation [1]. In the model, a program consists of a high-level map function and reduce function which process key-value pairs. If a problem is formulated in this way, it can be parallelized automatically by the MapReduce framework.

A MapReduce operation takes place in two main stages. In the first stage, the map function is called once for each input record. At each call, it may produce any number of output records. In the second stage, this intermediate output is sorted and grouped by key, and the reduce function is called once for each key. The reduce function is given all associated values for the key and outputs a new list of values (often "reduced" in length from the original list of values).

A *map function* is defined as a function that takes a single key-value pair and outputs a list of new key-value pairs. The input key may be of a different type than the output keys, and the input value may be of a different type than the output values:

$$\mathsf{map} : (K_1, V_1) \to \mathsf{list}((K_2, V_2))$$

A *reduce function* is a function that reads a key and a corresponding list of values and outputs a new list of values for that key. The input and output values are of the same type. Mathematically, this would be written:

$$\mathsf{reduce} : (K_2, \mathsf{list}(V_2)) \to \mathsf{list}(V_2)$$

Although the formal definition of map and reduce functions would indicate building up a list of outputs and then returning the list at the end, it is more convenient in practice to emit one element of the list at a time and return nothing. Conceptually, these emitted elements still constitute a list.

Figure 1 shows the task dependencies in a MapReduce operation. Since the map function only takes a single record, all map operations are independent of each other and fully parallelizable. A reduce operation may depend on the output from any number of map calls, so no reduce operation can begin until all map operations have completed. However, the reduce operations are independent of each other and may be run in parallel. The data given to map and reduce functions are generally fine-grained to ensure that the implementation can split up and distribute tasks. The MapReduce system consolidates the intermediate output from all of the map tasks. These records are sorted and grouped by key before being sent to the reduce tasks.

These map and reduce functions are sometimes deceptively simple. Even for applications which are simple on the surface, it is inherently difficult to implementing a scalable distributed system with fault-tolerance and load-balancing. In the MapReduce model all of this complexity is found in the surrounding MapReduce framework rather than in the map and reduce functions.

Although not all algorithms can be efficiently formulated in terms of map and reduce functions, MapReduce provides benefits over many other popular parallel processing systems. In this model, a program consists of only a map function and a reduce function. The infrastructure provided by a MapReduce implementation manages all of the details of communication, load balancing, fault tolerance, resource allocation, job startup, and file distribution. Those who write mappers and reducers can focus on the problem at hand without worrying about implementation details.

A more complex program may consist of multiple MapReduce stages combined together. In an iterative MapReduce program, the output of each reduce task is the input to a subsequent map task. Figure 2 shows the task dependencies in an iterative MapReduce operation. Iterative programs are more sensitive to the overhead of the MapReduce implementation. The per-iteration overhead is multiplied by the number of iterations, which can number in the tens or hundreds of thousands in some applications.

Most MapReduce implementations have targeted large-scale data processing, though a few systems have focused on improving performance for computationally intensive programs. Google has described some details of its MapReduce implementation in published papers and slides, but the system is private. The Apache Lucene project developed Hadoop, an Java-based open-source MapReduce framework implementation. Hadoop is the largest and most well known MapReduce implementation, and though it is primarily designed for large-scale data processing, it has also been used for computationally intensive tasks. HaLoop [6] is an example of a project intended to improve the performance of Hadoop for iterative programs. Twister [7] is an alternative MapReduce implementation designed to improve performance of iterative programs with some sacrifice of fault tolerance. Hadoop remains the most well known and widely available MapReduce system.

III. MapReduce in Scientific Computing

Most MapReduce systems are designed to operate in racks of computers dedicated to MapReduce processing, but scientific applications use a wide range of computing resources. A company such as Google or Yahoo with large datacenters might have many thousands of MapReduce systems and thousands of jobs per day, and dedicated systems in datacenters are appropriate in this situation. However, many different types of clusters are used for scientific computing. Shared clusters, which are generally large supercomputers, use batch scheduling systems to coordinate jobs submitted by a wide variety of users. Private clusters, consisting of a smaller number of commodity workstations or temporarily provisioned cloud nodes, are used by a single user at a time and do not require a scheduler. Shared and private clusters are different from dedicated MapReduce clusters. A shared cluster has many users, each of which has unique software requirements. Supercomputers provide resources that are expected to meet the needs of the majority of users, and any individual user cannot expect MapReduce infrastructure to be available. Likewise, most private clusters have no support staff to set up software. In many cases on both shared and private clusters, an individual MapReduce user must perform installation, configuration, and maintenance of the infrastructure they require.

Since MapReduce systems like Hadoop are designed to operate on dedicated machines, these frameworks implement functionality that may duplicate and conflict with the native equivalents that are provided on a supercomputer, such as a job scheduler and a custom distributed filesystem. Most MapReduce frameworks include a job scheduler, but shared clusters already provide a scheduler such as PBS, and private clusters may not require a scheduler at all. A redundant or unnecessary job scheduler does not introduce irreconcilable conflicts but does add complexity in configuration, maintenance, and running jobs. Likewise, a distributed filesystem is redundant with a supercomputer's high-availability centralized storage. Requiring the use of a particular distributed filesystem adds great complexity in maintenance and may be less robust in this context than the existing storage system. After all, a distributed filesystem expects nodes to be up all the time, but a supercomputer's scheduler kills processes as soon as a job completes. The distributed filesystem may lose all of its data nodes and all associated data within a few seconds. It is inappropriate for a MapReduce system on a supercomputer to use a specialized distributed filesystem or scheduler.

A MapReduce system that is well suited to scientific computing on supercomputers will meet different requirements than a MapReduce system designed for large-scale corporate data processing. Such a system should be easy to install and maintain and should play well with existing infrastructure. Jobs should be easy to submit to a batch scheduling system. Programs should require minimal boilerplate to allow for rapid development. Many scientific programs are dynamic research code rather than stable production software, so the system should make it easy to develop and debug programs on a single workstation or small cluster while scaling up to supercomputers.

The Python programming language presents both advantages and challenges in the context of scientific MapReduce. On the one hand, Python is a full-featured and popular language that maximizes developer productivity. It has a large scientific community, and it is popular for developing both prototypes and production code. It has a full-featured standard library, and most third-party libraries are easy to install in a home directory. Python interfaces with other languages, such as C, C++, and Java (through Jython). PyPy is an alternative Python interpreter that provides high performance, especially for numerical programs. On the other hand, as a highly dynamic language, it does not prioritize performance above all other concerns. However, with careful attention to the language's limitations, it is entirely possible to write an efficient MapReduce implementation in Python.

IV. The Design and Architecture of Mrs

Mrs is a lightweight MapReduce implementation that works well for scientific computing. It is designed to be simple for both programmers and users. The API includes reasonable but overridable defaults in order to avoid any unnecessary complexity. Likewise, Mrs makes it easy to run jobs without requiring a large amount of configuration. It supports both Python 2 and Python 3 and depends only on the standard library for maximum portability and ease of installation. Furthermore, Mrs is designed to easily run in a varienty of enviornments and filesystems. Mrs is also compatible with PyPy, a high-performance Python interpreter with a JIT compiler that accelerates numerical-intensive programs particularly well.

Mrs does not assume any particular job scheduler and is convenient to run in a variety of different contexts. Starting a job requires merely starting one copy of the program as a master and any number of other copies of the program as slaves. It does not require any running daemons, any configuration files, or any particular network ports. When the master starts, it writes its port to a file (unless a fixed port is specified). A slave needs only the master's address and port to connect. Scripts that automate the startup process are available both for shared clusters such as university supercomputers with many users and for private clusters with a small number of users. The script for shared clusters submits a job to a PBS queue (and is easily adapted for any other batch scheduler). The script for private clusters starts the master and uses pssh (parallel-ssh) to start slaves given a list of hosts. In all cases, configuration consists only of a short list of command-line options.

A. Programming Model

As a programming framework, Mrs controls the execution flow and is invoked by a call to `mrs.main`. The execution of Mrs depends on the command-line options and the specified program class. In its simplest form, a *program class* has an `__init__` method which takes the arguments `opts` and `args` from command-line parsing and a `run` method that takes a `job` argument. In practice, most program classes inherit from `mrs.MapReduce`, which provides a variety of reasonable but overridable defaults including `__init__` and `run` methods that are sufficient for many simple programs. The simplest MapReduce program need only implement a `map` and a `reduce` method.

Mrs provides several features to make writing, testing, and debugging MapReduce programs easier. First, it can run a program in several different execution contexts to help a programmer track down bugs. Second, it provides a simple mechanism for generating independent streams of pseudorandom numbers to make it easy to ensure that results are deterministic and repeatable. Third, it includes a specialized programming model for high-performance iterative MapReduce algorithms.

Mrs defines several different implementations which define the run-time behavior of a program. The *master/slave implementation* distributes work across a cluster of processors. The *serial implementation* performs all work sequentially on a single processor and makes all work deterministic. The *mock parallel implementation* splits work into the same tasks as would be run in the master/slave implementation but performs all computation on a single processor. Intermediate data between tasks is saved to files which can be helpful for debugging. The *bypass implementation* invokes the program class's optional `bypass` method, which is a simple entry point that avoids almost all of the functionality of Mrs. This implementation makes it easy to share code between a simple serial implementation of a program and the corresponding MapReduce implementation. A program's master/slave, serial, mock parallel, and bypass implementations should all produce identical answers, Differences in behavior between any two implementations, even in stochastic algorithms, indicate a bug in the program or possibly in Mrs.

Mrs provides a mechanism for defining independent streams of pseudorandom numbers. Nondeterministic results fundamentally make debugging difficult and testing impossible. In sequential programs, setting a random seed is a simple way to make stochastic algorithms deterministic. However, in a MapReduce program, setting a fixed random seed at the beginning of each map or reduce task would make all tasks use the same sequence of random numbers. The `mrs.MapReduce` class provides a `random` method that returns a random number generator. The method takes a variable number of integer arguments and ensures that the random number generator is unique for any particular combination of inputs. Because of the large size of the internal state of the Mersenne Twister, the `random` method can accept around 300 arguments that are each 64-bit integers. Mrs makes it easy to generate a unique random number generator in each task or even to create identical random number generators in different tasks that need to duplicate specific calculations.

Mrs is optimized for high-performance iterative algorithms. In most MapReduce systems, there is a significant delay between the end of one iteration and the beginning of the next. Between iterations, a program must retrieve results, check for convergence, and submit a new MapReduce job, which can involve a considerable amount of overhead. Mrs allows a program to queue up map and reduce operations so that each is ready to begin as soon as the previous operation finishes. It can also run operations in parallel if they do not depend on each other. For example, a convergence check can run in parallel with the computation of subsequent iterations. The task scheduler in Mrs also attempts to assign corresponding tasks to the same processor from one iteration to the next, which reduces communication between nodes and latency between iterations. While outside the scope of this work, Mrs includes several other optimizations to improve the performance of computationally intensive iterative algorithms. Support for iterative algorithms allows Mrs to efficiently run iterative algorithms that would otherwise be inappropriate for MapReduce.

B. Architecture

Mrs owes much of its efficiency to simple design. Many choices are driven by concerns such as simplicity and ease of maintainability. For example, Mrs uses XML-RPC because it is included in the Python standard library even though other protocols are more efficient. Profiling has helped to identify real bottlenecks and to avoid worrying about hypothetical ones. We include a few details about the architecture of Mrs.

Communication between the master and a slave occurs over a simple HTTP-based remote procedure call API using XML-RPC. Intermediate data between slaves uses either direct communication for high performance or storage on a filesystem for increased fault-tolerance. Mrs can read and write to any filesystem supported by the Linux kernel or FUSE, including NFS, Lustre, and the Hadoop Distributed File System (HDFS), and native support for WebHDFS is in progress. For data stored to a filesystem, the writer opens and writes a file and then sends the master the corresponding URL, which is used for any future reads. For data communicated directly, the writer opens and writes a file on a local filesystem, and requests from readers are served by a built-in HTTP server. Though direct communication writes to a local filesystem, small short-lived files are rarely written to disk. Rather, they stay in the kernel's filesystem buffer and are served and removed without ever being flushed.

Python requires a bit more attention to detail than some other languages to properly manage threads. Within each master and slave, Mrs generally uses processes instead of threads because of Python's threading model. The Python language specifies a Global Interpreter Lock (GIL) that prevents multiple threads in a single process from executing at the same time. Because of the GIL, Mrs uses threads sparingly, with their use limited to multiplexed I/O threads. Any threads must be started after all processes have started to avoid any risk of forking while holding a lock. In general, all child threads are configured as *daemon* threads, meaning that they are automatically terminated by Python when the main thread completes. This ensures that a straggling thread does not prevent the program from terminating. In each process, the main thread runs an event loop based on `poll`. Main threads do not wait on locks for extended periods of time because `wait` is not generally interruptible by signals including keyboard interrupts. To avoid such problems and to allow threads to wait on network communication and other threads at the same time, Mrs makes heavy use of pipes. Writing a single byte to a pipe wakes up `poll` in a remote process or thread and causes it to continue through its event loop. Communication between the processes of a master or slave uses Python's `multiprocessing` module which is also based on pipes. Complex Python programs like Mrs are much more robust and easily designed by making greater use of processes and pipes and only sparing use of threads and locks.

V. Evaluation

Our objective in creating Mrs was to make MapReduce programming fundamentally more accessible. We have sought to take full advantage of the features and facilities in Python to make Mrs both fast and easy to use. In this section we evaluate our results with the Mrs framework in two ways, first a subjective assessment of programming and running Mrs and second a quantitative assessment of performance and scalability. In both the subjective evaluation and the performance measurements, we will compare Mrs with Hadoop, which is currently the most popular MapReduce framework we know of.

A. Subjective Assessment

In this subsection we seek to assess how effective Mrs is for program development and for execution. While a software engineering-type comparative study (with multiple groups coding the same application under control circumstances) is outside of the scope of this paper, we present here what we feel is compelling subjective evidence that that Mrs is an easier environment the development of MapReduce programs.

The most well known MapReduce example is WordCount, a program which counts the number of occurrences of each word in a document or set of documents. This example problem comes from the original MapReduce paper [1]. For this program, the input and output sets, needed for MapReduce as defined in Section II, are:

$$\begin{aligned} K_1 &: \mathbb{N} \\ V_1 &: \text{set of all strings} \\ K_2 &: \text{set of all strings} \\ V_2 &: \mathbb{N} \end{aligned}$$

In WordCount, the input value is a line of text. The input key is ignored but generally arbitrarily set to be the line number for the input value. The output key is a word, and the output value is its count.

Program 1 shows the complete Mrs code for the WordCount example. The Mrs implementation follows trivially from the MapReduce approach to the problem described in that paper. The "map" part of the implementation splits the input line into individual words and emits one key-value pair for each word in the input with the word token serving as the key and a constant string representation of the number 1 as the value. In this application, this is the so-called "embarrassingly parallel" part of the program, separate processes can be dispatch to emit these key value pairs for each file or even parts of files without concern that they will conflict with each other.

The MapReduce framework groups all messages with matching keys, via a sort step. The framework passes the key and a list of all of the values with that key to the reduce part of the application. The reduce function in the WordCount example, also shown in Algorithm 1, takes a word (the key) and the list of counts, performs a sum reduction, and emits the result. This is the only element emitted, so the output of the reduce function is a list of size 1.

Although not critical to an understanding of MapReduce nor this example, the MapReduce architecture allows for an interesting optimization. If the map tasks emit a large number of records (as in WordCount), the sort step can take a long time. MapReduce addresses this potential problem by introducing the concept of a combiner function. If a combiner is available, the MapReduce system will locally sort the output from several map calls on the same machine and perform a "local reduce" using the combiner function. This reduces the amount of data that must be sent over the network for the main sort leading to the reduce phase. In WordCount, the reduce function can function as a combiner without any modifications. In our quantitative results included below, we make use of this optimization in both the Mrs version and the java version.

Program 2 shows the code for the same application but for the Hadoop framework (without the needed imports, to conserve space) taken from the examples included with Hadoop.

The same basic structure is discernible. In this case there is a class to hold the needed map and reduce functions. Java forces exception processing to be more visible than it was in the Python version. Likewise the marshalling of data is verbose. Some of the complexity of the main function is driven by the fact that Hadoop makes more of the job structure visible where as Mrs finds the needed elements through introspection. Likewise typing in java adds to the complexity of Hadoop programming. It is certainly the case that Hadoop requires users to know much about how the systems works. Where as Mrs really just needs the map and reduce functions which is the whole point of MapReduce programming.

One might assume that running a Mrs job would be more complex than running a Hadoop job because Mrs generally relies on external systems for job management and communications, Fortunately that is not the case. In the shared cluster context running a Mrs job is quite easy. Program 3 shows the basic elements of a PBS script for running a Mrs MapReduce program. This script and the corresponding Hadoop script represent have been reduced to show just the minimum script elements required to start MapReduce jobs. Full scripts include additional error handling and output specification. Any environment variables not defined within the scripts are assumed

Program 1 WordCount in Mrs/Python

```
import mrs

class WordCount(mrs.MapReduce):
    def map(self, key, value):
        for word in value.split():
            yield (word, 1)

    def reduce(self, key, values):
        yield sum(values)

if __name__ == '__main__':
    mrs.main(WordCount)
```

to be set externally.

The Mrs script (Program 3) has four basic parts: finding the network address of the master, starting the master, wait for the master to start, and starting the slaves.

The corresponding Hadoop (program 4) script has more issues to address because Hadoop was designed to run on dedicated hardware. When trying to simply run as mapReduce program, there are many elements that have to be setup and later shut down. There are 6 major part of this scripts. As with the mrs scripts, first the network address must me found. Second, the Hadoop configuration must be setup. Note that these files are oriented to the operations of a dedicated infrastructure, thus in some cases (just one in this case, but it could be worse) configuration files must be edited (see the "sed" line), not just moved into place. Next the daemon processes must be started on the master node (step 3) and so too must the daemons for with the slave nodes (step 4). Now the Master task for the MapReduce can be run (step 5). Lastly, in step 6, the daemons on both the master and slaves can be stopped. Note also that since Hadoop requires that data be stored in the Hadoop file system (HDFS) it must be created and formatted as part of this process, which was included as part of step 3. Furthermore, any data to be processed by the MapReduce program must be copied into the HDFS, and likewise data produced, but be copied back out before the HDFS is deleted. The copying of data in and out of the HDFS is accounted for in step 5. Again, in the context of a dedicated system, many of these steps are not needed but on a share cluster, they are.

B. Performance

In this section we will demonstrate how Mrs performance compares to that of Hadoop using three example problems of increasing relevance to scientific computing: WordCount, Pi, and Particle Swarm Function Optimization (PSO). For the first two of these experiments we used our private cluster of 21 machines, each with 6 cores. For the last set experiments involving empirical function optimization, we used the Fulton Supercomputing Lab at Brigham Young University. For all experiments using Hadoop we used the Hadoop file system

Program 2 WordCount in Hadoop (imports omitted)

```
public class WordCount {
 public static class TokenizerMapper
    extends Mapper<Object, Text, Text, IntWritable> {
  private final static IntWritable one = new IntWritable(1);
  private Text word = new Text();

  public void map(Object key, Text value, Context context
     ) throws IOException, InterruptedException {
   StringTokenizer itr =
      new StringTokenizer(value.toString());
   while (itr.hasMoreTokens()) {
    word.set(itr.nextToken());
    context.write(word, one);
   }
  }
 }
 public static class IntSumReducer
    extends Reducer<Text,IntWritable,Text,IntWritable> {
  private IntWritable result = new IntWritable();

  public void reduce(Text key,
     Iterable<IntWritable> values, Context context
     ) throws IOException, InterruptedException {
   int sum = 0;
   for (IntWritable val : values) {
    sum += val.get();
   }
   result.set(sum);
   context.write(key, result);
  }
 }
 public static void main(String[] args) throws Exception {
  Configuration conf = new Configuration();
  String[] otherArgs = new
     GenericOptionsParser(conf, args).getRemainingArgs();
  if (otherArgs.length != 2) {
   System.err.println("Usage: wordcount <in> <out>");
   System.exit(2);
  }
  Job job = new Job(conf, "word count");
  job.setJarByClass(WordCount.class);
  job.setMapperClass(TokenizerMapper.class);
  job.setCombinerClass(IntSumReducer.class);
  job.setReducerClass(IntSumReducer.class);
  job.setOutputKeyClass(Text.class);
  job.setOutputValueClass(IntWritable.class);
  FileInputFormat.addInputPath(job,
     new Path(otherArgs[0]));
  FileOutputFormat.setOutputPath(job,
     new Path(otherArgs[1]));
  System.exit(job.waitForCompletion(true) ? 0 : 1);
 }
}
```

Program 3 Mrs Startup Script (PBS)

```
# Step 1: Find the network address.
ADDR=$(/sbin/ip −o −4 addr list "$INTERFACE"
    |sed −e 's;^.∗inet \(.∗\)/.∗$;\1;')

# Step 2: Start the master.
PORT_FILE="$JOBDIR/port"
$PYTHON $MRS_PROGRAM −−mrs=Master \
    −−mrs−runfile="$PORT_FILE" ${ARGS[@]}

# Step 3: Wait for the master to start.
while [[ ! −e $PORT_FILE ]]; do sleep 1; done
PORT=$(cat $PORT_FILE)

# Step 4: Start the slaves.
pbsdsh bash −i −c "$PYTHON $MRS_PROGRAM
    −−mrs=Slave −−mrs−master='$ADDR:$PORT'"
```

Program 4 Hadoop Startup Script (PBS)

```
# Step 1: Find the network address.
ADDR=$(/sbin/ip −o −4 addr list "$INTERFACE"
    |sed −e 's;^.∗inet \(.∗\)/.∗$;\1;')

# Step 2: Set up the Hadoop configuration.
export HADOOP_LOG_DIR=$JOBDIR/log
mkdir $HADOOP_LOG_DIR
export HADOOP_CONF_DIR=$JOBDIR/conf
cp −R $HADOOP_HOME/conf $HADOOP_CONF_DIR
sed −e "s/MASTER_IP_ADDRESS/$ADDR/g"
    −e "s@HADOOP_TMP_DIR@$JOBDIR/tmp@g" \
    −e "s/MAP_TASKS/$MAP_TASKS/g" \
    −e "s/REDUCE_TASKS/$REDUCE_TASKS/g" \
    −e "s/TASKS_PER_NODE/$TASKS_PER_NODE/g" \
    <$HADOOP_HOME/conf/hadoop−site.xml \
    >$HADOOP_CONF_DIR/hadoop−site.xml

# Step 3: Start daemons on the master.
HADOOP="$HADOOP_HOME/bin/hadoop"
$HADOOP namenode −format # format the hdfs
$HADOOP_HOME/bin/hadoop−daemon.sh start namenode
$HADOOP_HOME/bin/hadoop−daemon.sh start jobtracker
```

(HDFS) since it is required. For the data intensive WordCount application we also used HDFS, but also tried NFS. The NFS results differ very little (about a second) from those reported. When using HDFS we dedicated one machine as the HDFS name node, Hadoop job tacker and Mrs master. We also assumed that the HDFS was already running for both frameworks. For Hadoop, we ensured that all Hadoop deamons and task trackers were already running. In this way, we measured the performance of the actual MapReduce programs, and not the infrastructure supporting the MapReduce frameworks.

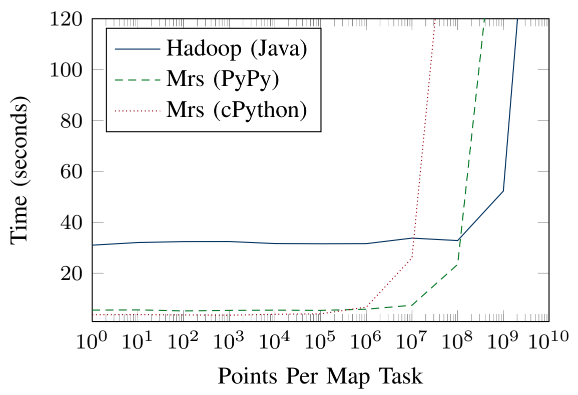
Our first example uses the WordCount problem, as this task is common in MapReduce literature. We use all of the text works from Project Gutenburg, a freely available collection of public domain ebooks (omitting files such as music or Human Genome Project data). Our full dataset of the works available in pure ASCII format includes 31,173 files, for a total of roughly two billion unique word tokens. We utilize HDFS to store this data for both Mrs and Hadoop.

Unfortunately for our comparison, the directory structure from Project Gutenburg is not very amenable to Hadoop. The input file loader for the Hadoop system expects all of the files to be located in a single directory, which is not the case with the Project Gutenburg dataset. With the full dataset, Hadoop struggles to load the data from so many locations, making the start up time alone take nearly nine minutes. In contrast, Mrs is able to perform the entire ReduceMap operation, which included loading the data, counting the words, and aggregating the counts, in under nine minutes. We feel that the directory structure of Project Gutenburg is representative of real world data and that fundamentally Mrs is more flexible in terms of loading data. Note that on a smaller subset of the data with only 8,316 files, Hadoop takes one minute to prepare the data, with a total time of sixteen minutes to finish, while Mrs finishes the entire MapReduce operation in just two minutes.

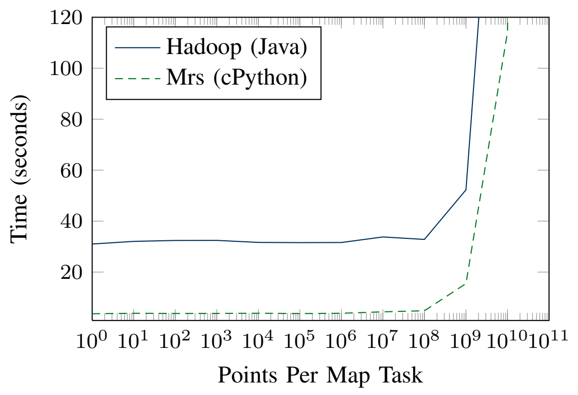
While WordCount is a canonical MapReduce example, the PiEstimator example in Hadoop is more representative of the numeric, computationally intensive problems encountered in most scientific computation. PiEstimator computes the value of π using a simple Monte Carlo method. While trivial to implement, this task is computational in nature, with no data on disk. This method consists of generating a large number of sample points uniformly distributed on a square with area of 1. An estimation of π / 4 is achieved by multiplying the ratio of points which fall within unit circle centered at a corner of the square to the total number of points. Multipling this value by 4 yields the final approximation of π.

PiEstimator generates random numbers using Halton sequences. While these sequences are entirely deterministic, they are quasi-random. Compared to uniform random numbers, Halton sequences tend to generate numbers which cover the sample space more evenly, which can lead to better results in certain types of Monte Carlo simulations. In all languages, the implementation of the Halton sequence is optimized to minimize the number of function calls and the number of comparison operations.

Figure 3 shows the results using Hadoop, Mrs with Python, Mrs with PyPy, and Mrs using `ctypes` to call a C function. We see two interesting trends. On the left-hand side of the graph, we see that Mrs significantly outperforms Hadoop, regardless of the choice of Python interpreter. This can be attributed to the high overhead inherent in using the Hadoop framework. For this problem, in human terms, it may not matter that the task completed in two seconds verses thirty seconds However, as we will discuss shortly, many scientific applications are iterative in nature, and this cost in overhead is multiplied by the number of iterations, making this a strong



(a) Halton sequence with Mrs using pure Python.



(b) Mrs with the inner loop implemented in C.

Fig. 3: Run times for estimating the value of π. The left hand side of the plots indicates that Mrs has significantly less overhead than Hadoop. The right hand side shows the performance of the numerical code, which is exponential due to the log scale. The algorithm is identical in all cases, so the differences reveal the performance penalty of each programming language.

advantage of the Mrs framework over Hadoop. As we look to the right hand side of Figure 3a, we see that the excellent numeric performance of Java begins to win out over pure Python. This can be attributed to the static nature of Java and the high quality of the Java JIT. While not unexpected, this does highlight a weakness of using pure Python for scientific computing.

However, Python makes it easy to rework existing code so that performance critical parts of an application, such as the inner loop of our map tasks, can be rewritten in C. For our second experiment approximating the value if π, we use Python's `ctypes` module to call a C function instead of the the pure Python implementation of the Halton sequence to uniformly generate random points. In this way we were able to very easily replace the inner loop of our map task with optimized C code, while leaving the rest of the loop unchanged. Figure 3b shows the results. Once again we see on the left that Mrs has extremely low overhead compared

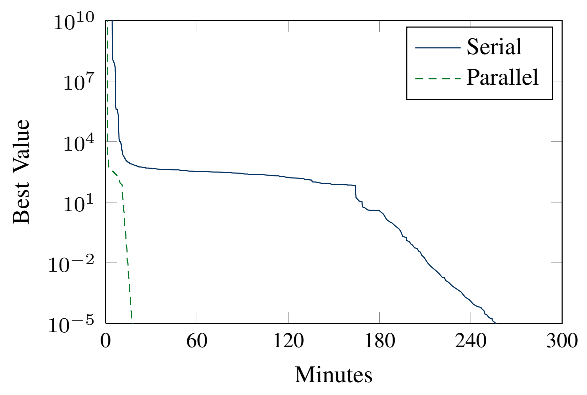


Fig. 4: Convergence plots of the Apiary topology for the Rosenbrock-250 function with respect to function evaluations and time.

to Hadoop. However, the C function is much faster than the corresponding Java function, so Mrs is much faster than Hadoop, despite the vast majority of Mrs code being in Python.

This experiment does show a key advantage of Python over other languages like Java—Python is designed to easily interface with other languages. We assert that the speed of Python will rarely be the true source of performance problems in the Mrs framework. Instead, thoughtful consideration of algorithms, coupled with profiling and careful optimization will yield the most improvement. In essence, Python allows us to quickly implement scientific application code, and then easily convert any critical paths to C. Java on the other hand, suffers somewhat in this respect.

Our final experiment is an optimization technique used in actual scientific computing. Particle Swarm Optimization (PSO) is an empirical function optimization algorithm inspired by simulations of flocking behaviors in birds and insects [8], [9]. The algorithm simulates the motion of a set of interacting particles within a multidimensional space. At each iteration, a particle moves and evaluates the objective function at its new position. A particle is drawn toward the best value it has seen and the best value that any of its neighbors has seen. PSO can be naturally expressed as a MapReduce program, with the map function performing motion simulation and evaluation of the objective function and the reduce function calculating the neighborhood best by combining the updated particle with messages from its neighbors [5]. For computationally trivial objective functions, task granularity can be too fine if each map task operates on a single particle. In this case, a swarm can be divided into several subswarms or islands, and each map task operates on several iterations of a subswarm of particles [10]–[12].

Using the "Apiary" approach for subswarming [12], Figure 4 shows the results for the well-know Rosenbrock benchmark function in 250 dimensions ("Rosenbrock-250") with both serial and parallel computation. Performing 100 iterations on 5 particles requires only 0.2 seconds, and parallel PSO took about 0.5 seconds per iteration. Note that Mrs took advantage of the the features for iterative MapReduce that we previously mentioned. Furthurmore, this figure includes only the overhead between iterations, and not the start up time for Mrs (which is about 2 seconds). With any realistically expensive function, the overhead of 0.3 seconds would be negligible.

While we did not actually run PSO using Hadoop, we can estimate its performance. From the execution in Mrs, we know that for the Rosenbrock-250 function PSO took an average of 2471 iterations to reach the target value of 10^{-5}. From our experiments with calculating π, we know that Hadoop takes approximately 30 seconds per iteration. Thus Hadoop would take approximately $2471 * 30$ seconds or a little longer than 20 hours to achieve the same convergence. While we realized that this figure is a rough estimate, our experience with Hadoop suggest that for iterative tasks of this nature, the overhead of Hadoop often makes it slower than running the same task in serial on a single machine.

VI. Conclusion

The concept of MapReduce has allowed many users to express their scientific computations in an easily parallelizable way. However, the complexity of existing MapReduce frameworks often presents a significant programming burden. Furthermore, most existing MapReduce frameworks such as Hadoop are optimized for performing high volume data analysis rather than solve numerically intense problems. We have presented Mrs as a MapReduce framework which not only eases the programming barrier to entry, but is highly efficient in a scientific context.

Mrs is particularly well suited to an academic or research environment. Many universities and research institutions have supercomputer clusters, or private clusters but these are often generic in nature, not tied to any particular problem or parallel processing technology. Existing frameworks such as Hadoop which require a dedicated cluster and extensive configuration are not always suitable for researchers. Mrs on the other hand has proven exceptionally easy to install and use in a wide variety of environments, scheduling systems, and filesystems.

The choice of Python as our implementation language also aids researchers. Python is a language which naturally lends itself to readable and maintainable code. The syntax is clear and powerful, allowing users to quickly develop, test and deploy scientific applications. Furthermore, Mrs itself takes advantage of Python to make writing MapReduce programs easier. As we have demonstrated, Python lends itself to optimization, without sacrificing code quality by allowing bottleneck portions to be converted to C without affecting any other Python code.

Furthermore, the performance characteristics of Mrs are tailored for scientific computing, where overhead can be more of a significant issue. In particular, the low overhead of Mrs has improved our ability to tackle iterative evaluations such as empirical function optimization. While it was beyond the scope of this paper, future work on Mrs will include additional features which will further improve iterative MapReduce. We have developed a model for iterative MapReduce which allows for efficient implementation. In addition, we have developed a novel operations, which lower overhead by significantly reduce the need for communication compared to traditional MapReduce systems.

References

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proc. Operating System Design and Implementation*, 2004.

[2] W. Zhao, H. Ma, and Q. He, "Parallel k-means clustering based on MapReduce," *Cloud Computing*, 2009.

[3] C. Chu, S. Kim, Y. Lin, Y. Yu, G. Bradski, A. Ng, and K. Olukotun, "Map-Reduce for machine learning on multicore," in *Proc. Advances in Neural Information Processing Systems*, 2007.

[4] C. Jin, C. Vecchiola, and R. Buyya, "MRPGA: an extension of MapReduce for parallelizing genetic algorithms," in *IEEE International Conference on eScience*, 2008.

[5] A. McNabb, C. Monson, and K. Seppi, "MRPSO: MapReduce particle swarm optimization," in *Proc. Conference on Genetic and Evolutionary Computation*, 2007.

[6] Y. Bu, B. Howe, M. Balazinska, and M. Ernst, "HaLoop: Efficient iterative data processing on large clusters," *Proc. VLDB Endowment*, vol. 3, no. 1-2, 2010.

[7] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S. Bae, J. Qiu, and G. Fox, "Twister: a runtime for iterative MapReduce," in *Proc: High Performance Distributed Computing*, 2010.

[8] J. Kennedy and R. C. Eberhart, "Particle swarm optimization," in *Proc. International Conference on Neural Networks IV*, 1995.

[9] D. Bratton and J. Kennedy, "Defining a standard for particle swarm optimization," in *Proc. IEEE Swarm Intelligence Symposium*, 2007.

[10] J. Schutte, J. Reinbolt, B. Fregly, R. Haftka, and A. George, "Parallel global optimization with the particle swarm algorithm," *International Journal for Numerical Methods in Engineering*, vol. 61, no. 13, 2004.

[11] J. Romero and C. Cotta, "Optimization by island-structured decentralized particle swarms," in *Proc. Fuzzy Days: Computational Intelligence, Theory and Applications*, 2005.

[12] A. McNabb and K. Seppi, "The apiary topology: Emergent behavior in communities of particle swarms," *Parallel Problem Solving from Nature*, 2012.