RABID: A Distributed Parallel R for Large Datasets

Hao Lin
Electrical and Computer Engineering
Purdue University
West Lafayette, IN
Email: haolin@purdue.edu

Shuo Yang
R&D Center
Huawei Technologies
Santa Clara, CA
Email: shuo.yang@huawei.com

Samuel P. Midkiff
Electrical and Computer Engineering
Purdue University
West Lafayette, IN
Email: smidkiff@purdue.edu

Abstract—Large-scale data mining and deep data analysis are increasingly important for both enterprise and scientific applications. Statistical languages provide rich functionality and ease of use for data analysis and modeling and have a large user base. R[5] is one of the most widely used of these languages, but is limited to a single threaded execution model and problem sizes that fit in a single node. This paper describes highly parallel R system called RABID (R Analytics for Big Data) that maintains R compatibility, leverages the MapReduce-like distributed Spark[22] and achieves high performance and scaling across clusters. Our experimental evaluation shows that RABID performs up to 5x faster than Hadoop and 20x faster than RHIPE on two data mining applications.

Keywords—Distributed Computing, Big Data analytics, R, Data Mining

I. INTRODUCTION

Commercial, scientific and engineering applications have led to very large datasets becoming common. Logging of data for security and Web 2.0 apps and scientific and engineering computing all generate large datasets. For example, CERN generates one petabyte of data every second and stores 24 petabytes of data each year which must be analyzed and interrogated.

Domain experts are interested in what can be discovered from data, not learning new programming languages, and so providing parallel computing tools that are compatible with widely used languages is important. R is the top software tool in the data analytics community[17], and its popularity is growing. R, as a sequential language, is easy to use but is limited to single core performance and the memory on a single node. An R execution engine that allows users to easily execute their R programs on parallel systems with high performance would allow analysts to efficiently target large datasets without the distraction and pain of learning a new language. RABID is such a system.

Other efforts have targeted parallel R execution[16]. Snow[20] and Snowfall[13] are built on MPI but only provide limited parallel APIs. The pbdMPI package[18] requires an R user to use low-level message-passing calls, a major break from the R programming model. Its extension, pbdDMat, provides a higher level distributed matrix abstraction based on MPI but primarily targets linear algebra. None of these MPI-based systems support fault tolerance.

RABID is an R system that allows R users to implement efficient analyses (both iterative and non-iterative) of datasets on parallel systems without needing to learn new programming models, languages or parallel frameworks. RABID’s combination of high-performance and R-compatibility places it in the upper right corner of the chart in Figure 1. RABID accomplishes this in three ways.
First, RABID is implemented on the Spark framework. Spark outperform Hadoop by 20x[22] on iterative jobs and provides fault tolerance and high availability, which RABID uses. This alone is insufficient for great parallel R performance. Second, RABID uses distributed data structures that act like regular R data structures and a serialization strategy that is transparent to users and is compatible with the Renjin[2] R execution engine utilized by RABID. This enables R compatibility and reduces the memory footprint of a RABID job. Third, R performs optimizations that reduce the communication overhead using three strategies. RABID uses a static analysis to determine the subset of data visible within a function that is actually needed by the function, reducing the communication volume of data sent to an R function, and the memory footprint by 80% of the original. Next, RABID performs operation merging to dramatically reduce the communication of intermediate values by 60% on average. Finally, RABID uses communication pipelining to allow communication and computation to proceed in parallel, and increase performance by 20% on average in our applications. These techniques together allow RABID to support the familiar R programming model while providing users with high performance parallel executions.

To summarize, this paper makes the following contributions:

1) It describes the RABID system, its distributed data structures, supported operations and runtime system that allow R code to scale to distributed systems.

2) It describes RABID’s distributed data structures and serialization techniques that enable parallel execution within the R programming model.

3) It describes optimizations to reduce communication overheads and techniques to reduce the memory footprint, allowing larger problems to be run.

4) It shows RABID’s performance on two benchmarks relative to RHtPE and Hadoop, and has speedups of up to 20x and 10x, respectively.

This paper is organized as follows. Section II describes the R and RABID programming model and RABID’s architecture. Design decisions and the analyses and optimizations performed by RABID to provide superior parallel performance, while maintaining the R programming model, are described in Section III. In Section IV we describe our benchmark programs and the experimental evaluation of the system. Finally Sections V and VI describe related work and conclusions.

II. PROGRAMMING IN R AND RABID

Before discussing how RABID supports the R programming model, it is useful to review the R programming model by means of an example.

A. The R Programming Model

R is, at its core, a functional language that passes data through a series of operators (often represented as functions) that repeatedly transform the data. Iterative computations often involve an invariant input data set that is transformed by a sequence of operations within a loop. The result of the operations is the input to the next iteration of the loop. Operators that transform an input data D into D' is called the execution trace of D'. Loop bodies often form a trace.

Figure 3 shows an R implementation of the K-means clustering algorithm. Line numbers are shown for clarity and are not part of the program. The RABID API lapply() (line 1) applies the R function as.numeric to each line read by rb.readLines() and as.numeric() converts each line to a numeric vector. The cache parameter requests that R cache the working set of vectors in RAM if possible. Then DATA in a list type is transformed into a data frame (line 2). Next (line 4), a set of randomly picked centroids is defined as a list. In the iterative optimization loop (lines 6–15), both adply() (line 9) and aggregate() , a reduce-like function that groups records by user-specified keys, shown in line 8, are repeatedly invoked to compute the centroid closest to each record held in DATA. The user defined function func and built-in function mean update centroids with the mean of all points in the same cluster, i.e., aggregated with the same key id. The loop continues until the convergence condition is reached. We note that the trace for DATA of the result of each loop iteration is the two adply and one aggregate operation.

B. Supporting the R Programming Model in RABID

The RABID system allows high performance parallel executions of R on clusters. It does this by bridging the gap between R and the parallel data engine. By targeting Spark we can leverage its efficiency in handling iterative computations and its fault tolerance mechanisms.

Figure 4 shows the RABID software stack. RABID provides distributed data structures and low-level and high-

```r
1. DATA <- lapply(rb.readLines('hdfs://. . .') as.numeric(a), cache=T)
2. DATA <- as.data.frame(DATA, row.names=...)
3. set.seed(1000)
4. centroids <- as.list(sample(DATA, 16))
5. func <- function(a) { ... }
6. while(cond < threshold) {
7. newCen <- as.list(aggregate(  
8. x=adply(DATA, 1, func),
9. by=id,  
10. FUN=mean))
11. tempDist <- sum(mapply(function(x,y) dist(x[2],y[2]),
12. centroids, newCen))
13. centroids <- newCen)
14. centroids <- newCen
15. }
```

Figure 3. K-means clustering as an example of using RABID. Vanilla R API functions overloaded in RABID are in bold font, RABID distributed data structures are in all capital letters.
level operations on those data structures. The low-level operations target RABID’s distributed R list and the high-level operations target distributed data frames and matrices as well as providing parallel data mining functions. We also support access to the Hadoop Distributed File System (HDFS) from R programs. The R task coordinates distributed R code hosted by either the Renjin or GNU R interpreters on server nodes and the optimizer and scheduler, as the name implies, performs optimizations described in Section III and schedules distributed R functions.

We now discuss in more detail the distributed functions, data types and APIs supported by RABID that are essential to the RABID’s R compatibility.

1) Distributed Functions: In the example of Figure 3, the RABID versions of `lapply()` and `aggregate()` are data parallel and `as.list()` is used to collect distributed data into an R list on the master machine. Except for `rb.readLines()`, all other RABID functions override standard R and provide users with signatures that differ only in optional parameters. In the signatures, we also preserve the R ‘...’ argument so that a user passes in additional argument settings to functions like `lapply()` and `aggregate()`.

2) Distributed Data Types and APIs: RABID provides several R-compatible data types that are “drop-in” replacements for the corresponding standard R data types. A key data type for R is the list. Unlike arrays, R lists are heterogeneous, i.e., members can have different types. RABID supports the BigList distributed type which is transparent to users and can be used exactly like the standard R list data type. RABID also provides distributed R data types such as matrices and data frames, which contain well-structured data with rows of records and columns of attributes, similar to database data tables.

Each distributed dataset is accessed as an instance of a “Big*” class. These objects are descriptors, or promise objects, i.e., instead of actually containing the data they store the necessary information to obtain the data. Thus distributed datasets are lazily evaluated and operations are executed only when their values are needed, which provides opportunities for performance optimizations.

More low-level APIs are described in [15]. Table I shows high-level RABID APIs that manipulate the BigMatrix and BigDataFrame data types. Aside from taking optional tuning parameters to control communication granularity they are identical to vanilla R APIs. Higher-level APIs are mainly implemented in R by using low-level APIs.

C. The RABID Runtime Architecture

The RABID runtime architecture is shown in Figure 5. R scripts, written by a RABID user, are submitted through a web server to the RABID (and Spark) master that runs the user’s command. An R session process (R driver) on the master runs the user’s script with RABID support. It keeps dataset variables in symbol tables, schedules DAG structured jobs[22] and maintains user defined R functions (UDFs). By default, the R code is executed using the Renjin[2] R virtual machine, which is written in Java. Renjin was chosen because it, like Spark, is implemented in Java, and consequently can be better integrated with Spark.

To enable communication of R data types (represented as Java objects in Renjin) to different servers, RABID rewrites these objects as Java `Serializable` objects. The serialization of `AtomicVector` data types, e.g. integer, character and logical vectors, is straightforward as only the primitive data fields are serialized. For `Compound Vectors`, e.g., the `ListVector` that is essentially an array of R objects, the serialized object consists of a length followed by serialized element objects. Most importantly, new user defined data types in R are automatically serializable as they are compositions of basic R types.

In both R and RABID, computation using both built-in functions and UDFs are represented in the system as `Function` objects, i.e., a closure object. A closure in R is a composition of formal parameters and a function body as pair-lists, along with the closure’s enclosing environment. Enclosing environments can be divided into two categories: user created environments and default R environments (i.e. `namespaces` in R default core packages). Since the core packages are installed at startup on servers the default R environments need not be serialized.

On the server node side (see Figure 5), the R interpreter can carry out computations in two different ways. By default, R tasks on worker processes are executed by the Renjin R interpreter and share the copy of the dataset that is cached by the Spark Java worker process. We refer to this default mode as `shared-mem` mode. As we show in Section III, `shared-mem` mode reduces both latency and memory overheads. Renjin is capable of running all existing R packages by rebuilding them, and the Renjin community is working to translate R packages, but this effort is currently incomplete. When a RABID user tries to load an R library that has not been rebuilt, RABID uses the C-based GNU R interpreter and the `data streaming` mode instead of the `shared-mem` mode, as shown in Figure 5. In the `data streaming` mode,
each RABID operation execution (1) starts a Spark operation that connects the tasks with a set of long running R session daemons through sockets; (2) starts a pipelined data transmission; and (3) creates a new distributed dataset for the output. Data and messages are transferred back and forth through these sockets. Data serialization and data transfers are costly and data is duplicated as there is a copy cached by Spark and another used by GNU R processes.

A. Reducing the Memory Footprint

There are two causes of excessive memory pressure in R. The first results from R being a dynamically scoped language, i.e., free variables in a function are resolved to the variable with the same name in the environment of the nearest calling function on the runtime stack. Free variables in UDFs to be remotely executed must be identified and added to the environment sent to the remote node. Simply serializing and sending the enclosing dynamic environments to the remote node would communicate large amounts of unneeded data. RABID uses Algorithm 1 to determine the free variables used by a UDF and bind the UDF to the new environment. The algorithm recursively visits the child nodes of UDF’s Abstract Syntax Tree (AST) looking for variables accessed in the UDF body that are not parameters. These variables and their values are added into the new UDF’s environment and sent with the UDF. For better performance UDFs are then compiled with this environment into byte code by the R “compiler” package[5]. R processes for this UDF are created on the remote nodes and the serialized code and environment are sent to these processes.

The second cause of excessive memory pressure is when R packages not rebuilt for Renjin are used. RABID must transport data in the R format to the underlying Spark server and then make it available to the GNU R interpreter. This extra copy of the data significantly increases the memory pressure.

B. Optimizing Communication

1) Pipelining Data to R Sessions: RABID supports pipelined communication between Spark and R workers to overlap communication and computation. The underlying Spark HDFS breaks data into splits, where a split is the unit of data assigned to a mapper. To support pipelining, RABID divides splits into smaller data blocks, allowing computation to begin when the first data block arrives at a worker. RABID provides a default size for data blocks and provides an optional user parameter to specify the block size. We see that performance is stable over a large range of data block sizes. We also note that breaking data into splits allows less data to be present on a node at any given time, further reducing memory pressure on the node.

As seen in Fig. 6, RABID reads records from a file one-by-one. Map-like operations (e.g. `lapply()`) are applied to a record and the result is stored into a fixed sized R list

---

**Table 1**

<table>
<thead>
<tr>
<th>BigDataFrame</th>
<th>BigMatrix</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rb.data.frame()</code> creates a distributed &quot;data frame&quot; dataset</td>
<td><code>rb.matrix(...)</code> creates a distributed matrix dataset</td>
</tr>
<tr>
<td><code>$</code> returns a BigList column of specified attribute</td>
<td><code>$</code> more matrix and statistical computation</td>
</tr>
<tr>
<td><code>aggregate(x, by, FUN)</code></td>
<td><code>+</code>, <code>-</code>, <code>*</code>, <code>/</code>, <code>%%</code>, <code>^</code>, <code>%%</code> basic matrix arithmetic</td>
</tr>
</tbody>
</table>

**III. MAKING RABID EFFICIENT: RABID OPTIMIZATIONS**

Five challenges faced by any parallel and distributed R systems are (1) Maintaining the R programming model; (2) Enabling efficient iterative parallel R execution; (3) Providing fault tolerance; (4) Minimizing the memory footprint and (5) Minimizing communication overheads during the execution of an R program. The first challenge is to maintain usability and RABID’s solution was described in Section II. RABID meets the second and third challenges by being implemented on top of the Spark framework, which allows for efficient execution of iterative computations and provides fault tolerance. In this section we first discuss how RABID meets the fourth and fifth challenges of reducing the memory footprint and communication overheads. We then finish with a discussion of RABID’s fault tolerance mechanism that minimize the cost of failures.
Algorithm 1 Finding the data used by a function (closure).

**Input:** $n$: an expression node in the R AST, $f$: the UDF, $e$: new UDF environment to be serialized

**Algorithm:**

1. If $n$ has no children, then return $e$.
2. For $i = 1, 2, \ldots$, number of node $n$'s children
   - $\text{closure}(i)(n, f, e)$ /*recursive over children nodes*/
3. Else if $n$ has no child
   - If $n$ is NOT a variable identifier
     - Add $n$ into the UDF environment $e$

---

that is written to Spark when full and then refilled until the computation ends. For aggregate-like operations, records with the same user specified keys (specified by the “by” parameter) are also blocked into fixed sized R lists. Then the user keys are hashed into indices to map the records onto servers. The fixed size R list is further blocked into lists with the same server indices and the records are communicated to the specified server.

2) Distributing Computation to R Processes: Because system functions are already placed on server nodes, only UDFs need to be distributed. RABID uses the technique described in Section III-B2 to reduce the volume of data transmitted by only sending data from the environment that is needed by a function. For better performance, UDFs are then compiled with the identified environment into byte code by the R compiler package[5]. R processes for this UDF are created on the remote nodes, and the serialized code and environment are sent to these processes. RABID datasets are promise objects and therefore data accesses by the UDF begin the pipelined communication of needed data to the node and R process. This allows data transmission and UDF execution to be overlapped.

3) Merging Operations to Reduce Communication Overhead: The communication overhead can be reduced by merging a sequence of RABID operations that are in a trace, i.e., operations that consume data produced by earlier operations in the sequence, into a single Spark operation. We call these merged operations or m-ops. RABID merges adjacent operations that work on the same data split as long as there is no data shuffling or reconstruction of splits, as happens with aggregations. Thus aggregation-like operations are treated like barriers. Other operations, such as lapply() and Reduce() that precede or follow a barrier, can be merged into an m-op.

Merging operations provides significant benefits for data streaming workers. First, the merger allows one set of R processes to be created for the entire m-op rather than one for each operation within the m-op. Second, when data is pipelined to the m-op, one set of input data is transmitted to the m-op and all other data needed by constituent operations is produced within the m-op. This is much cheaper than creating a new distributed dataset on Spark and writing results to it with redundant data transmission and serialization, as would be necessary if the operations of the m-op were executed separately. With m-ops there is a single UDF transmission, a single pipelined communication phase and startup, and a single output distributed dataset created and written.

The storage information held by an R promise object includes boolean variables indicating if the data object is materialized, i.e. if it is cached or checkpointed. When a dataset needs to be materialized the R driver process will track its trace back to a materialized ancestor dataset. Starting from this dataset down to the current dataset to be materialized, all of the UDFs with their environments in the trace are merged and all are passed to R worker processes on the server nodes. In this way, operations before the materialized data will not be recomputed.

After merging operations, RABID examines adjacent m-ops to determine if they share the same input data set. If they share the same input dataset the data need not be passed along with later functions as it will still be contained by the R daemon. This situation, illustrated in Figure 7, occurs frequently in some iterative algorithms that do not perform aggregations, such as Expectation Management (EM) algorithms of which K-means and LR are examples. The kernel of these algorithms is often a loop body consisting of several map-like or reduce-like operations with iteration and merged into a single m-op. We can observe that the m-op in each iteration takes the same input dataset and only differs in the computation, i.e. the function, which contains the statistical model refined by the previous iteration. We show the performance benefits of m-ops in Section IV.

C. Fault Tolerance

We take advantage of Spark’s existing heartbeat-based worker-side fault tolerance mechanisms to handle node failures. Tasks are restarted when the heartbeat indicates a worker task failure. Trace information for tasks stored in both the R driver process and the Spark master process allows lost results to be recomputed, reducing checkpointing overheads. Periodic checkpointing will still allow the system to recover faster from failures.
User code errors terminate R worker sessions. In RABID, these errors will be caught and immediately terminate the job (fast fail) so that Spark’s fault-tolerance mechanism will not try and recompute tasks or other useless operations. RABID also collects R’s verbose error information from each worker’s stderr to give users information about the cause of a failure. On the R master, job information such as the symbol table of dataset variables, program execution state and worker information are backed up using Apache Zookeeper.

IV. Evaluation

A. Experimental Setup

We use two workloads, an R implementation of logistic regression (which we refer to as LR) that implements a gradient descent algorithm to compute the weight of 1 billion 10-D data points and an R implementation of K-means that performs a clustering on movie ratings. Both are widely used in data mining applications. Our experiments show the R scripts running on RABID provide improved performance compared to the implementations in Hadoop 0.20.205 and RHIPE 0.7. First, we show that RABID scales well with these workloads and that in the data streaming mode our optimizations give significantly performance improvements.

LR and K-means are both iterative algorithms. LR is run using a synthetic dataset with 1 billion 10-D data points and an R implementation of K-means that performs a clustering on movie ratings. Both are widely used in data mining applications. Our experiments show the R scripts running on RABID provide improved performance compared to the implementations in Hadoop 0.20.205 and RHIPE 0.7. First, we show that RABID scales well with these workloads and that in the data streaming mode our optimizations give significantly performance improvements.

LR and K-means are both iterative algorithms. LR is run using a synthetic dataset with 1 billion 10-D data points and an R implementation of K-means that performs a clustering on movie ratings. Both are widely used in data mining applications. Our experiments show the R scripts running on RABID provide improved performance compared to the implementations in Hadoop 0.20.205 and RHIPE 0.7. First, we show that RABID scales well with these workloads and that in the data streaming mode our optimizations give significantly performance improvements.

B. Experimental Results

Figure 8 shows our evaluation of RABID performance on LR and K-means. Figure 8(a) shows the execution time of our approach compared to the Hadoop and RHIPE implementations of LR on datasets with 1 million, 10 million, 100 million and 1 billion points over 3 iterations. RABID outperforms RHIPE by an order of magnitude and Hadoop by a smaller amount. Once the data are cached in the first iteration, later iterations are performed more quickly by in-memory computation in RABID. Thus RABID gives better speedups than Hadoop and RHIPE when multiple iterations are needed. This effect is shown in Figure 8(b), which compares the time to complete the first and later iterations. Figure 8(c) shows scalability over an increasing number of nodes, with an input dataset of 1 billion points. There is a super-linear speedup going from 10 to 18 nodes in RABID because the input dataset can be completely cached in memory in 18 nodes but not 10.

Figure 8(d) shows the time spent for runs of K-means with an input movie dataset of 100 million, 300 million, 1000 million and 3000 million ratings over 3 iterations on a 26-node cluster. RABID again shows significant performance benefits compared to Hadoop and RHIPE. Figure 8(e) show that as with LR, RABID’s performance increases in later iterations. Figure 8(f) shows the scalability over the number of nodes, with an input dataset of 1 billion points.

We also observed that RABID workers in shared memory mode consume only 55% of the memory of RHIPE workers. When the dataset volume increases, RABID performance benefits more because more partitions of the dataset are able to be cached in memory. In contrast, R worker processes in frameworks like RHIPE use more memory.

C. M-ops in Data Streaming Workers

In Logistic Regression, adjacent iterations transform the same input dataset with different UDFs. If data streaming to workers is used, RABID caches the dataset in the R worker daemons to be used in the next iteration. Figure 9(a) shows the effect of using this feature of operation merging with dataset reuse enabled. As seen in the Figure, this optimization saves a great deal of time spent in data serialization and transmission. Since most of the linear algebra computation is done in native libraries, the performance can be even better than the shared-memory worker model that is using interpreted execution. As seen in Figure 9(b) data streaming uses extra memory.

We now examine how the data block size affects the performance in streaming workers. Figure 9(c) gives the runtime with various block sizes and it can be seen that aside from very small block sizes performance is not affected by the block size.

Figure 9(d) presents the size of closures that are transmitted to workers when Algorithm 1 is and is not used, and shows that the algorithm reduces the volume of data transferred. We note that as the program executes the amount of data in variables can grow dramatically.

V. Related Work

We now discuss related work not covered in the introduction.

The Hadoop[1] ecosystem (MapReduce[9], Distributed File System (HDFS), etc.) and Dryad[12] are widely used.
SystemML[11] is a matrix-based Hadoop extension framework for machine learning that uses a domain specific language. Hadoop’s design, however, precludes it from efficiently solving iterative problems because of job setup overheads and unnecessary disk I/Os[7, 22] and it requires R users to learn to program in another system.

HaLoop[7] is a variation of Hadoop optimized to run to run iterative MapReduce jobs and Twister[10] has an API that is similar to HaLoop’s. All of these frameworks require programming against Java-based APIs instead of R.

MLlib[14] is a set of distributed low-level machine learning libraries written in Scala using the Spark runtime. MLlib[19] is a set of higher level APIs providing machine learning programming abstractions. Our work can be viewed as a framework that is as powerful as MLlib and MLI combined and easier to use by traditional R programmers. RABID provides a framework for R programmers to develop distributed machine learning algorithms without learning a new DSL a a new general-purpose such as programming language Java or Scala.

SystemML[11] is a matrix-based Hadoop extension framework for machine learning using an R-like DSL (Domain Specific
Language). The DSL approach also aims to provide a better user interface than the Hadoop APIs. SystemML differs from our approach both in being a DSL, not being R, and in not implementing distributed data types such as lists and data frames. RABID, being compatible with R, can utilize all vanilla R features, which SystemML cannot.

Presto[21] is a newly developed distributed R system that targets sparse matrix computation. Some commercial solutions such as RevoScaleR[3], from Revolution Computing seem to leverage Hadoop for parallel R execution but details of their approach are not available in the literature.

VI. CONCLUSIONS AND FUTURE WORK

RABID provides R users with a familiar programming model that scales to large clusters, allowing larger problem sizes to be efficiently handled. Unlike other systems that require R programmers to use unfamiliar languages or programming models, RABID users can write R scripts to create a data analysis job. RABID is implemented on Spark and uses operation merging, data pipelining and analysis of the environment variables needed by a UDF to further improve performance. RABID outperforms Hadoop and RHIPE on our benchmarks. Development continues on RABID to support more high-level functions and to implement further optimizations. RABID is cloud-ready and future work will target cloud systems.

REFERENCES