Distributed R for Big Data

Indrajit Roy
HP Vertica Development Team

Abstract

Distributed R simplifies large-scale analysis. It extends R. R is a single-threaded environment which limits its utility for Big Data analytics. Distributed R allows users to write programs that are executed in a distributed fashion, that is, parts of programs as specified by the developer can be run in multiple single-threaded R-processes. The result is dramatically reduced execution times for Big Data analysis. This tutorial explains how Distributed R language primitives should be used to implement distributed analytics.

Keywords: R, distributed execution, `darray`, `foreach`.

1. Introduction

Many applications need to perform advanced analytics such as machine learning, graph processing, and statistical analysis on large-amounts of data. While R has many advanced analytics packages, the single-threaded nature of the R limits their use on Big Data. Distributed R extends R in two directions:

- **Distributed data.** Distributed R stores data across servers and manages distributed computation. Users can run their programs on very large datasets (such as Terabytes) by simply adding more servers.
- **Parallel execution.** Programmers can use Distributed R to implement code that runs in parallel. Users can leverage a single multi-core machine or a cluster of machines to obtain dramatic improvement in application performance.

Distributed R provides distributed data-structures to store in-memory data across multiple machines. These data-structures include distributed arrays (`darray`), distributed data-frames (`dframe`), and distributed lists (`dlist`). These data structures can be partitioned by rows, columns, or blocks. Users specify the size of the initial partitions. Distributed arrays should be used whenever data contains *only* numeric values. Distributed data-frames should be used for non-numeric data. Distributed lists can store complex objects such as R models.

Programmers can express parallel processing in Distributed R using `foreach` loops. Such loops execute a function in parallel on multiple machines. Programmers pass parts of the distributed data to these functions. The Distributed R runtime intelligently schedules functions on remote machines to reduce data movement.

In addition to the above language constructs, Distributed R also has other helper functions. For example, `distributedR_start` starts the Distributed R runtime on a cluster. Information about all the functions is present in the Distributed R Manual. It is also available via the `help()` command in the R console. Unlike the manual, the focus of this tutorial is to show,

through examples, how Distributed R functions are used to write analytics algorithms.

2. Distributed R architecture

Before explaining the Distributed R programming model, it is important to understand the system architecture. Distributed R consists of a single *master* process and multiple *workers*. Logically, each worker resides on one server. The master controls each worker and can be co-located with a worker or started on a separate server. Each worker manages multiple local R instances. Figure 1 shows an example cluster setup with two servers. The master process runs on server A and a worker runs on each server. Each worker has three R instances. Note that you could setup the workers to use more or fewer R instances.

For programmers, the master is the R console on which Distributed R is loaded using `library(distributedR)`. The master starts the program and is in charge of overall execution. Parts of the program, those corresponding to parallel sections such as `foreach`, are executed by the workers. Distributed data structures such as `darray` and `dframe` contain data that is stored across workers. The Distributed R API provides commands to move data between workers as well as between master and workers. Programmers need not know on which worker data resides, as the runtime hides the complexity of data movement.

3. Programming model

Distributed R is R with new language extensions and a distributed runtime. Distributed R contains the following three groups of commands. Details about each command can be obtained by using `help` on each command or by reading the Distributed R Manual.

Session management:

- `distributedR_start` - start session
- `distributedR_shutdown` - end session
- `distributedR_status` - obtain master and worker information

Distributed data structures:

- `darray` - create distributed array
- `dframe` - create distributed data frame
- `dlist` - create distributed list
- `as.darray` - create darray object from matrix object
- `npartitions` - obtain total number of partitions
- `getpartition` - fetch darray, dframe, or dlist object
- `partitionsize` - size of partition
- `clone` - clone or deep copy of a darray

Parallel execution:

- `foreach` - execute function on cluster
- `splits` - pass partition to foreach loop
- `update` - make partition changes inside foreach loop globally visible

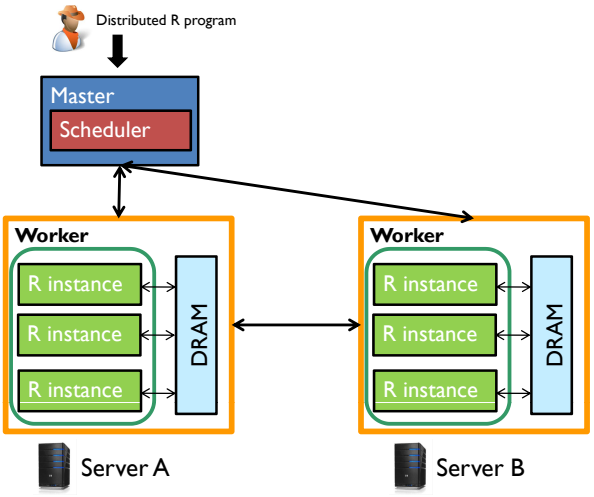

Figure 1: Distributed R architecture.

3.1. Distributed data-structures

Distributed arrays (`darray`) provide a shared, in-memory view of multi-dimensional data stored across multiple servers. Distributed arrays have the following characteristics:

- **Partitioned.** Distributed arrays can be partitioned into contiguous ranges of rows, columns, or blocks. Users specify the size of the initial partitions. Distributed R workers store partitions of the distributed array in the compressed sparse column format unless the array is defined as dense. Programmers use partitions to specify coarse-grained parallelism by writing functions that execute in parallel and operate on partitions. For example, partitions of a distributed array can be loaded in parallel from data stores such as HP Vertica or from files. Programmers can use `getpartition` to fetch a distributed array and materialize it at the master node. For example, `getpartition(A)` will reconstruct the whole array `A` at the master by fetching the partitions from local and remote workers. The i^{th} partition can be fetched by `getpartition(A,i)`.

- **Shared.** Distributed arrays can be read-shared by multiple concurrent tasks. The user simply passes the array partitions as arguments to many concurrent tasks. However, Distributed R supports only a single writer per partition.

A distributed data frame (`dframe`) is similar to a `darray`. The primary difference is that, unlike `darray`, distributed data frames can store non-numeric data. Even though a `dframe` can be used to store numeric only data, it is much more efficient to use `darray` in such cases. The efficiency difference is because of the underlying representation of these data structures.

Distributed list (`dlist`) stores elements inside lists that are partitioned across servers. To create a distributed list, programmers only need to specify the number of partitions. For example, `dlist(5)` will create a distributed list with five partitions. Initially each partition is a R list with no elements.

3.2. Parallel programming

Programmers use `foreach` loops to execute functions in parallel. Programmers can pass data, including partitions of `darray` and `dframe`, to the functions. Array and data frame partitions can be referred to by the `splits` function. The `splits` function automatically fetches remote partitions and combines them to form a local array. For example, if `splits(A)` is an argument to a function executing on a worker then the whole array `A` would be re-constructed by the runtime, from local and remote partitions, and passed to that worker. The i^{th} partition can be referenced by `splits(A,i)`.

Functions inside `foreach` do not return data. Instead, programmers call `update` inside the function to make distributed array or data frame changes globally visible. The Distributed R runtime starts tasks on worker nodes for parallel execution of the loop body. By default, there is a barrier at the end of the loop to ensure all tasks finish before statements after the loop are executed.

4. Examples

We illustrate the Distributed R programming model by discussing a number of examples.

4.1. Getting started

Follow the steps in the installation guide to first install Distributed R. Load the Distributed R library and then start the cluster by calling `distributedR_start`.

```
> library(distributedR)
> distributedR_start()

Master address:port - 127.0.0.1:50000
```

You can view the status of the cluster with `distributedR_status`. It shows details such as the number of workers in the cluster, number of R instances managed by each worker, system memory available on each worker node, and so on.

```
> distributedR_status()

          Workers Inst SysMem MemUsed DarrayQuota DarrayUsed
1 127.0.0.1:50001    4  15759    2855        7091          0

> distributedR_shutdown()
```

The last command shuts down the Distributed R cluster.

4.2. Creating a distributed array

Next, create a distributed array.

Create a 9x9 dense array by specifying its size and how it is partitioned. The example below shows how to partition the array into 3x3 blocks and set all its elements to the value 10. Therefore, there are 9 partitions that could reside on remote nodes.

```
> library(distributedR)
> distributedR_start()

Master address:port - 127.0.0.1:50000

> A <- darray(dim=c(9,9), blocks=c(3,3), sparse=FALSE, data=10)
```

You can print the number of partitions using `npartitions` and fetch the whole array at the master by calling `getpartition`. If you have a really large array, such as one with billions of rows, fetching the whole array at the master is not a good idea as it defeats the purpose of managing huge datasets by distributing data across multiple workers.

```
> npartitions(A)

[1] 9

> getpartition(A)

     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
[1,]   10   10   10   10   10   10   10   10   10
[2,]   10   10   10   10   10   10   10   10   10
[3,]   10   10   10   10   10   10   10   10   10
[4,]   10   10   10   10   10   10   10   10   10
[5,]   10   10   10   10   10   10   10   10   10
[6,]   10   10   10   10   10   10   10   10   10
[7,]   10   10   10   10   10   10   10   10   10
[8,]   10   10   10   10   10   10   10   10   10
[9,]   10   10   10   10   10   10   10   10   10
```

Typically, you partition arrays by rows or columns (i.e., 1-D partitioning) instead of blocks (i.e., 2-D partitioning). Since row and column partitioning is a special case of block partitioning, this example details block partitioning. If you partition the array `A` by rows by using `blocks=c(3,9)` instead of `blocks=c(3,3)`, then each partition will contain 3 rows and all the columns.

Distributed arrays can initially be declared empty. For example, it is typical to create an array and then load data into the array from a data store. The initial declaration will create a full array which is soon overwritten. By declaring an array empty, you can save memory space.

```
> Aempty <- darray(dim=c(9,9), blocks=c(3,3), sparse=FALSE, empty=TRUE)
> npartitions(Aempty)

[1] 9

> getpartition(Aempty,1)
```

```
<0 x 0 matrix>
```

4.3. Parallel programming with `foreach`

The `foreach` loop is a flexible and powerful construct to manipulate distributed data structures. This example illustrates its use by initializing a distributed array with different values.

Create another distributed array `B` with the same size (9x9) as `A` and partitioned in the same manner. In our previous example, we used the argument `data` to initialize all elements of `A` to 10. However, you cannot use `data` to set different values to array elements. Instead, start a `foreach` loop, pass partitions of `B`, and inside the loop assign values to the partition.

```
> B <- darray(dim=c(9,9), blocks=c(3,3), sparse=FALSE)
> foreach(i, 1:npartitions(B),
+ init<-function(b = splits(B,i), index=i){
+   b <- matrix(index, nrow=nrow(b),ncol=ncol(b))
+   update(b)
+ })
```

The syntax of `foreach` is `foreach(iteration variable, range, function)`. In the above example, `i` is the iteration variable which takes values from 1 to 9. Therefore, 9 parallel tasks are created that execute the functions. In the function, we pass the i^{th} partition of B using `splits(B,i)`. We also pass the value of the `i`. Within the function we assign a matrix to the partition. The matrix is of the same size as the partition (3x3) but initialized by the value of the iteration variable. This means that the i^{th} partition will have all elements equal to `i`. We can fetch the whole array by using `getpartition`.

```
> getpartition(B)
```

```
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
[1,]    1    1    1    2    2    2    3    3    3
[2,]    1    1    1    2    2    2    3    3    3
[3,]    1    1    1    2    2    2    3    3    3
[4,]    4    4    4    5    5    5    6    6    6
[5,]    4    4    4    5    5    5    6    6    6
[6,]    4    4    4    5    5    5    6    6    6
[7,]    7    7    7    8    8    8    9    9    9
[8,]    7    7    7    8    8    8    9    9    9
[9,]    7    7    7    8    8    8    9    9    9
```

A particular partition, say the 5^{th}, can be fetched by specifying the partition index.

```
> getpartition(B,5)
```

```
     [,1] [,2] [,3]
[1,]    5    5    5
[2,]    5    5    5
[3,]    5    5    5
```

There are few things to keep in mind while using `foreach`. First, only variables passed as arguments to the function (`init` in this case) are available for use within the function. For example, the array `A` or its partitions cannot be used within the function. Even the iterator variable (`i`) needs to be passed as an argument. Second, loop functions don't return any value. The only way to make data modifications visible is to call `update` on the partition. In addition, `update` can be used *only* on distributed data-structure (`darray`, `dframe`, `dlist`) arguments. For example, `update(index)` is incorrect code as `index` is not a distributed object.

4.4. Parallel array addition

With the two initialized distributed arrays, you can start computations such as adding their elements. We will again use a `foreach` loop to perform the parallel addition. First create an output array `C` of the same size and partitioning scheme. In the `foreach` loop pass the i^{th} partition of all three arrays, `A`, `B`, and `C`. Within the loop we add the corresponding partitions, put the output in `c`, and call `update`:

```
> C <- darray(dim=c(9,9), blocks=c(3,3))
> foreach(i, 1:npartitions(A),
+ add<-function(a = splits(A,i), b = splits(B,i), c = splits(C,i)){
+   c <- a + b
+   update(c)
+ })
> getpartition(C)
```

```
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
 [1,]   11   11   11   12   12   12   13   13   13
 [2,]   11   11   11   12   12   12   13   13   13
 [3,]   11   11   11   12   12   12   13   13   13
 [4,]   14   14   14   15   15   15   16   16   16
 [5,]   14   14   14   15   15   15   16   16   16
 [6,]   14   14   14   15   15   15   16   16   16
 [7,]   17   17   17   18   18   18   19   19   19
 [8,]   17   17   17   18   18   18   19   19   19
 [9,]   17   17   17   18   18   18   19   19   19
```

While `foreach` can be used to perform any parallel operation, Distributed R package provide basic operators that work out-of-the-box on distributed arrays. These operators include array addition, subtraction, multiplication, and summary statistics such as max, min, mean, and sum (including their column and row versions such as `colSums`). Internally, all these operators are implemented using `foreach`. The example below illustrates some of these operators in action:

```
> D <- A+B
> getpartition(D)
```

```
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
 [1,]   11   11   11   12   12   12   13   13   13
```

```
 [2,]   11   11   11   12   12   12   13   13   13
 [3,]   11   11   11   12   12   12   13   13   13
 [4,]   14   14   14   15   15   15   16   16   16
 [5,]   14   14   14   15   15   15   16   16   16
 [6,]   14   14   14   15   15   15   16   16   16
 [7,]   17   17   17   18   18   18   19   19   19
 [8,]   17   17   17   18   18   18   19   19   19
 [9,]   17   17   17   18   18   18   19   19   19
```

```
> mean(D)

[1] 15

> colSums(D)

[1] 126 126 126 135 135 135 144 144 144
```

4.5. Distributed array with flexible partition sizes

You can also create distributed arrays by specifying just the number of partitions, but not their sizes. This flexibility is useful when the size of an array is not known apriori. For example, create a distributed array with 5 partitions as follows:

```
> fA <- darray(npartitions=c(5,1))
```

Each partition can contain any number of rows and columns as long as it results in a well formed array. For example, you can store `i` rows in the ith partition.

```
> foreach(i, 1:npartitions(fA), initArrays<-function(y=splits(fA,i), index=i) {
+      y<-matrix(index, nrow=index,ncol=5)
+      update(y)
+ })
```

Check partition sizes by calling:

```
> partitionsize(fA)

     [,1] [,2]
[1,]    1    5
[2,]    2    5
[3,]    3    5
[4,]    4    5
[5,]    5    5
```

Contents of the second partition are:

```
> getpartition(fA,2)

     [,1] [,2] [,3] [,4] [,5]
[1,]    2    2    2    2    2
[2,]    2    2    2    2    2
```

Contents of the full array are:

```
> getpartition(fA)

      [,1] [,2] [,3] [,4] [,5]
 [1,]    1    1    1    1    1
 [2,]    2    2    2    2    2
 [3,]    2    2    2    2    2
 [4,]    3    3    3    3    3
 [5,]    3    3    3    3    3
 [6,]    3    3    3    3    3
 [7,]    4    4    4    4    4
 [8,]    4    4    4    4    4
 [9,]    4    4    4    4    4
[10,]    4    4    4    4    4
[11,]    5    5    5    5    5
[12,]    5    5    5    5    5
[13,]    5    5    5    5    5
[14,]    5    5    5    5    5
[15,]    5    5    5    5    5
```

4.6. Creating a distributed data frame

The syntax for distributed data frames is similar to distributed arrays. However, data frames can store non-numeric values.

Create a 9x9 data frame by specifying its size and how it is partitioned:

```
> dF <- dframe(dim=c(9,9), blocks=c(3,3))
```

The dataframe `dF` has 9 partitions each of size 3x3. Unlike, distributed arrays, the data frame has no elements unless data is explicitly loaded:

```
> getpartition(dF)

  X1 X2 X3 X1 X2 X3 X1 X2 X3
1  0  0  0  0  0  0  0  0  0
2  0  0  0  0  0  0  0  0  0
3  0  0  0  0  0  0  0  0  0
4  0  0  0  0  0  0  0  0  0
5  0  0  0  0  0  0  0  0  0
```

```
6  0  0  0  0  0  0  0  0  0
7  0  0  0  0  0  0  0  0  0
8  0  0  0  0  0  0  0  0  0
9  0  0  0  0  0  0  0  0  0
```

To add data, use a `foreach` loop:

```
> foreach(i, 1:npartitions(dF),
+ init<-function(df = splits(dF,i), index=i, n=3){
+     p <- matrix(index, nrow=n,ncol=n-1)
+     q <- rep("HP",n)
+     df<- data.frame(p,q)
+     update(df)
+ })
```

Each partitions now has a column which contains the string `HP`:

```
> getpartition(dF,1)

  X1 X2  q
1  1  1 HP
2  1  1 HP
3  1  1 HP
```

4.7. Creating a distributed list

Create a distributed list by specifying the number of partitions.

```
> dL <- dlist(partitions=3)
```

Affiliation:

Indrajit Roy
HP Vertica Development Team
URL: http://www.vertica.com/distributedr