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 runtime to manage distributed execution. 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
- `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

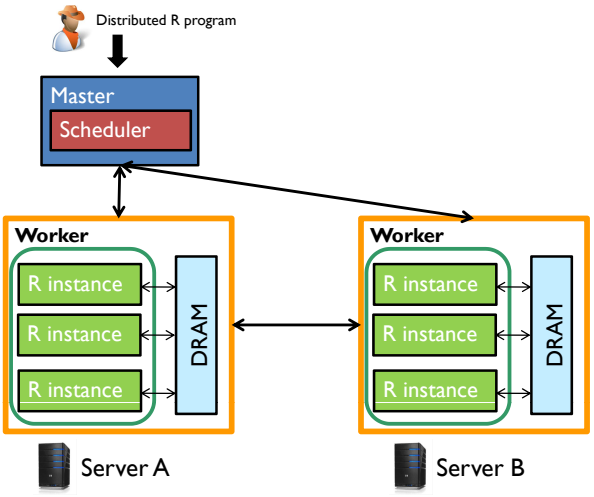

Figure 1: Distributed R architecture.

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 - 127.0.0.1:64967
[1] TRUE
```

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
1 127.0.0.1:57633    7   7806    3340        3512          0

> distributedR_shutdown()

[1] TRUE
```

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 - 127.0.0.1:53240
[1] TRUE

> 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.

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)
+ })

[1] TRUE
```

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)
+ })

[1] TRUE

> 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. 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)
```

```
data frame with 0 columns and 0 rows
```

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)
+ })
```

```
[1] TRUE
```

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
```

4.6. Creating a distributed list

Create a distributed list by specifying the number of partitions.

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

Initially, the list is empty.

```
> getpartition(dL)

list()
```

The list can be populated using the foreach loop.

```
> foreach(i, 1:npartitions(dL), function(dl=splits(dL,i), idx=i){
+     dl<-list(c("HP", idx))
+     update(dl)
+ })

[1] TRUE
```

Individual partitions or the whole list can be obtained by:

```
> getpartition(dL,1)

[[1]]
[1] "HP" "1"

> getpartition(dL)

[[1]]
[1] "HP" "1"

[[2]]
[1] "HP" "2"

[[3]]
[1] "HP" "3"
```

4.7. Load and save data from files

You can save or load data in parallel. Distributed R can run on top of databases such as HP Vertica and even file systems. Therefore, data can be loaded or saved to different stores as long as the right connector exists. Let's start with an example of saving data to files. Use the `foreach` loop to write each partition of an array to a file:

```
> fname <- paste(getwd(),"/Data",sep="")
> foreach(i, 1:npartitions(D),
+ saves<-function(d = splits(D,i), index=i, name=fname){
+     write(d, paste(name,index,sep=""))
+ })

[1] TRUE
```

The code above writes each partition in a different file. If Distributed R is running on a single machine, all the files are present on the same machine. You can load one of the partitions and check its contents:

```
> scan(paste(fname,5,sep=""))

[1] 15 15 15 15 15 15 15 15 15
```

Note that the above command may not work if Distributed R is running on a cluster as the file may be in a remote machine. Instead, use a `foreach` loop to load data in parallel into a distributed array. First declare a new distributed array `E`, of the same size as `D`, and then load data from previously saved files. Since `scan` returns the values as a single vector, first convert the data into a matrix for the correct size before calling `update`:

```
> E <- darray(dim=c(9,9), blocks=c(3,3))
> foreach(i, 1:npartitions(E),
+ loads<-function(e = splits(E,i), index=i, name=fname){
+     fn <- paste(name,index,sep="")
+     e <- matrix(scan(file=fn), nrow=nrow(e))
+     update(e)
+ })

[1] TRUE

> getpartition(E,5)

     [,1] [,2] [,3]
[1,]   15   15   15
[2,]   15   15   15
[3,]   15   15   15
```

4.8. Load and save data from HP Vertica

To load data from a database into a distributed array, use an ODBC connector such as HP Vertica RODBC (vRODBC) or vanilla RODBC. The example below shows how to load data using a `foreach` loop that makes concurrent ODBC connections to HP Vertica database. Declare a 50x4 array in which each partition contains 5 rows. Within the loop load each partition by querying the database for 5 rows at a time. Note that for this example to work, `vRODBC` needs to be installed and set up correctly to connect to HP Vertica database. Follow installation instructions of `vRODBC`. To use `RODBC`, just replace the occurrences of `vRODBC` with `RODBC` in the example below.

```
> X <- darray(dim=c(50, 4), blocks=c(5, 4), sparse=FALSE)
> foreach(i, 1:npartitions(X), initArrays <- function(x = splits(X,i), index=i) {
+  library(vRODBC)
+  connect<-odbcConnect("Test")
+  size <- nrow(x)
+  start <- (index-1) * size
+  end <- index * size
+  qry <- paste("select A,B,C,D from T where id >=", start,"and id <", end, "order by id")
+  segment<-sqlQuery(connect, qry)
+  odbcClose(connect)
+  x<-cbind(segment$A, segment$B, segment$C, segment$D)
+  update(x)
+ })

progress: 100%
[1] TRUE
```

There are two things to observe in this example. First, the programmer has to load the ODBC package inside the loop (using `library(vRODBC)`). This is necessary because the function inside the `foreach` loop executes on the worker and packages need to be explicitly loaded in the worker environment. Second, in this particular example we use HP Vertica's internal row identifiers to select rows. For example, the first 5 rows in the Vertica table `T` will be assigned to the first partition of array `X`. HP Vertica has row identifiers to refer to individual rows.

To fetch the first partition and display data, use `getpartition`:

```
> getpartition(X, 1)

     [,1]     [,2]     [,3]     [,4]
[1,]    5 0.903815 0.522466 0.250464
[2,]    1 0.994233 0.138644 0.139464
[3,]    3 0.117651 0.285975 0.309341
[4,]    4 0.280725 0.006694 0.684827
[5,]    6 0.331704 0.835160 0.498040
```

4.9. Parallel execution using existing packages

Sometimes, a problem can be solved by applying, in parallel, functions from existing packages. Take the example of finding the shortest distance from five source vertices to all other vertices in the graph. Since distance calculation from each source vertex is independent from others, we can start five tasks to calculate them in parallel. R already has a package called `igraph` that can calculate shortest distances. The example below details how to reuse `igraph` to solve the above problem. For this example, `igraph` needs to be installed on all machines in Distributed R cluster. You can manually download the software from CRAN or use `install.packages(igraph)`.

First, create a sparse distributed array to store the graph. Since we don't want to partition the graph, the array has only one partition equal to the total size of the graph.

```
> G<-darray(dim=c(100,100), blocks=c(100,100), sparse=TRUE)
```

Next, use a foreach loop to generate a random graph and store it in the array. Note that we need to load the `igraph` library inside the loop function.

```
> foreach(i, 1:1, initGraph<-function(g=splits(G)){
+  library(igraph)
+  rg<-erdos.renyi.game(nrow(g),0.1)
+  g<-get.adjacency(rg, sparse=TRUE)
+  update(g)
+ })

[1] TRUE
```

Now run parallel tasks to calculate shortest distances and store them in another array called `paths`. Partition the array `paths` such that each partition has one row and 100 columns. Therefore, each element in the array corresponds to the distance of the source vertex to a given destination vertex.

```
> paths<-darray(dim=c(5,100), blocks=c(1,100), sparse=FALSE)
> foreach(i, 1:npartitions(paths),
+ calc<-function(g=splits(G), p=splits(paths,i), vertex=i){
+  library(igraph)
+  p<-shortest.paths(graph.adjacency(g), vertex)
+  update(p)
+ })

[1] TRUE
```

Fetch all shortest distances from the first vertex and then print the first ten values:

```
> getpartition(paths, 1)[,1:10]

 [1] 0 1 3 3 2 2 2 1 2 2

> distributedR_shutdown()

[1] TRUE
```

5. Debugging distributed programs

Distributed R makes it easier to write distributed machine learning and graph algorithms. However, programmers may face challenges in debugging applications. If the error occurs in the sequential part of the code, i.e. outside the `foreach` loop, then Distributed R will typically display usual R error messages. You should use standard R techniques to debug the

program. The challenging part is if errors are in the parallel section of the code, i.e. related to the `foreach` loop. These errors may manifest on remote machines when functions are executed.

Here are few suggestions for debugging errors during parallel execution:

- **Check logs.** On each machine, the logs of the worker and each of the R instances are available under `/tmp/R_worker*` and `/tmp/R_executor*`. If Distributed R does not give a meaningful error message, the programmer may log into cluster machines and check the log files.
- **Add print statements.** If the logs don't contain enough information, we recommend the old school technique of adding lots of print statements in your code. Print statements inside the `foreach` loop will show up in `/tmp/R_executor*` logs.
- **Execute in local mode.** It is easier to debug programs on a single machine than a cluster. Since many errors in cluster mode will also manifest in a single multi-core machine, we recommend running Distributed R on a single machine using the same program and dataset. To further ease debugging, run Distributed R with only a single R instance by using `distributedR_start(inst=1)`.

6. Performance optimizations

When writing your own algorithms in Distributed R, the following considerations may help improve performance:

R instances. It is typical to use number of R instances on each worker machine equal to the number of cores on that machine. Starting more R instances than the number of cores may not improve performance as most algorithms are compute intensive. Using fewer cores may leave the machine underutilized.

Memory requirement. Distributed R is an in-memory system and requires all data to fit in the aggregate main memory of the cluster. In addition, you may need to have spare capacity for temporary data generated by the algorithm. For example, if the input dataset is 50GB and machines have 32GB RAM, you will need at least two such machines in the Distributed R cluster.

Array partitions. Each array partition is worked upon by an R instance. For good performance, number of partitions in a `darray` or `dframe` should be at least the number of R instances used. Otherwise, some R instances may remain idle. It is best to create arrays or data-frames with number of partitions equal to the total number of R instances or a *small multiple* of it. Too many partitions (e.g., more than 10× the number of total R instances) may degrade performance due to higher scheduling overheads. For example, consider an input dataset of size 50GB and a Distributed R cluster of two workers each running 5 R instances. Since there are a total of 10 R instances, create a distributed array with 10 (or 20 or 30) partitions to ensure each R instance is kept busy, load is evenly distributed, and scheduling overheads are low. When using 10 partitions, each partition will contain 5GB of data.

Using `getpartition`. The `getpartition` function fetches data from remote locations to the master node. It should not be used to fetch full arrays (or their partitions) which may not fit in the memory of the master node. In the previous example of a 50GB array distributed across two 32GB machines, `getpartition` on the full array will fail. Whenever possible, data should be manipulated in parallel using `foreach` instead of first moving data from remote nodes to master node with `getpartition`, and then applying the function on the single threaded master node. For example, it is better to calculate `mean` on a remote node using `foreach` instead of fetching the data on the master node and applying `mean`.

Running with HP Vertica. Since data mining algorithms are compute intensive, we recommend running Distributed R on a cluster of its own. If Distributed R and HP Vertica are run on the same servers, performance of both Distributed R and HP Vertica queries may get affected due to resource contention.

Affiliation:

Indrajit Roy
HP Vertica Development Team
URL: http://www.hpl.hp.com/distributedR.htm