User Guide for ddR

Edward Ma, Indrajit Roy, Michael Lawrence

2015-10-22

The 'ddR' package aims to provide an unified R interface for writing parallel and distributed applications. Our goal is to ensure that R programs written using the 'ddR' API work across different distributed backends, therefore, reducing the effort required by users to understand and program on different backend infrastructures. Currently 'ddR' programs can be executed on R's default 'parallel' package as well as the open source HP Distributed R. We plan to add support for SparkR. This package is an outcome of feedback and collaboration across different companies and R-core members!

'ddR' is an API, and includes a default execution engine, to express and execute distributed applications. Users can declare distributed objects (i.e., `dlist`, `dframe`, `darray`), and execute parallel operations on these data structures using R-style `apply` functions. It also allows different backends (that support ddR, and have ddR "drivers" written for them) to be dynamically activated in the R user's environment to execute applications

To get started, first install the 'ddR' package using `install.packages("ddR")`. Next, load the library. By default the package will use R's 'parallel' package as the backend. If you'd like to use a custom backend you will need to first install that backend and the 'ddR' driver for that backend. For example, if you'd like to use HP Distributed R, you will need to first install the `distributedR` package followed by the the driver `distributedR.ddR`.

```
library(ddR)
```

```
##
## Welcome to 'ddR' (Distributed Data-structures in R)!
## For more information, visit: https://github.com/vertica/ddR
##
## Attaching package: 'ddR'
##
## The following objects are masked from 'package:base':
##
##     cbind, rbind
```

```
## Run the next two lines also to use the Distributed R backend
# library(distributedR.ddR)
# useBackend(distributedR)
```

By default, the `parallel` backend is used with all the cores present on the machine. You can switch backends or specify the number of cores to use with the `useBackend` function. For example, you can specify that the `parallel` backend should be used with only 4 cores by executing `useBackend(parallel, executors=4)`. If you want to use SNOW based backend (also exported by `parallel`), you can set the `type` field to use sockets: `useBackend(parallel, type="PSOCK", executors=4)`. This command will launch four R processes that use sockets to communicate with each other. If the type field is not used, then the default is to fork multiple processes (not available on Windows).

Creating distributed objects

There are two ways to create distributed objects in 'ddR'.

1. Using the constructor functions.
2. Using `dmapply` or `dlapply` (`dlists` only).

Using constructor functions

A distributed list (`dlist`) may be created using the constructor with a comma-separated list of arguments. For example:

```r
my.dlist <- dlist(1,2,3,4,5)
my.dlist
```

```
##
## ddR Distributed Object
## Type: dlist
## # of partitions: 5
## Partitions per dimension: 5x1
## Partition sizes: [1], [1], [1], [1], [1]
## Length: 5
## Backend: parallel
```

Note that the output shows you a lot of useful information, including metadata about the object, partition sizes, etc. There are 5 partitions in `my.dlist`, because by default, the constructor creates as many partitions as the elements in input.

However, you may also specify the partitioning directly using the constructor:

```r
my.dlist <- dlist(1,2,3,4,5,nparts=3)
my.dlist
```

```
##
## ddR Distributed Object
## Type: dlist
## # of partitions: 3
## Partitions per dimension: 3x1
## Partition sizes: [2], [2], [1]
## Length: 5
## Backend: parallel
```

When `nparts` is supplied, it will create the requested number of partitions in the output object, with a best effort splitting of data between those partitions. In this case, since 5 isn't divisible by 3, it divides data into groups of 2, 2, and 1.

The constructors for `darray` and `dframe` are different. For example, you may initialize a `darray` in the following manner:

```r
my.darray <- darray(dim=c(4,4),psize=c(2,2),data=3)
my.darray
```

```
##
## ddR Distributed Object
## Type: darray
## # of partitions: 4
```

```
## Partitions per dimension: 2x2
## Partition sizes: [2, 2], [2, 2], [2, 2], [2, 2]
## Dim: 4,4
## Backend: parallel
```

This makes `my.darray` a `darray` that is filled with 3. Each partition is of size 2x2, and the dimension of `my.darray` is 4x4. For `dframe`, the constructor has the same format.

Using `dmapply` or `dlapply`

Constructors are handy when you want to initialize distributed objects quickly. However, the flexible way to create distributed objects is via `dmapply` and `dlapply`. Similar to the functional-programming behavior of R's mapply and lapply, distributed functions in 'ddR' also return new objects.

Below is an example on how to create the same `dlist` as before but by using `dlapply`:

```
my.dlist <- dlapply(1:5,function(x) x)
my.dlist
```

```
##
## ddR Distributed Object
## Type: dlist
## # of partitions: 5
## Partitions per dimension: 5x1
## Partition sizes: [1], [1], [1], [1], [1]
## Length: 5
## Backend: parallel
```

Why does it work? The distributed `lapply` function, `dlapply`, executes on the function argument vector `1:5`, assigning to each element the value of the vector for that iteration. To specify `nparts`, you can also supply `nparts`, just as in the constructor function:

```
my.dlist <- dlapply(1:5,function(x) x, nparts=3)
my.dlist
```

```
##
## ddR Distributed Object
## Type: dlist
## # of partitions: 3
## Partitions per dimension: 3x1
## Partition sizes: [2], [2], [1]
## Length: 5
## Backend: parallel
```

The behavior of `dmapply` for `darray` and `dframe` is slightly involved, though not substantially so. When creating these data structures using `dmapply`, the API needs to know some information, such as how to partition the output in 2d-manner, as well as how to combine intermediate results of dmapply within each partition. Therefore, you may need to also supply arguments for the following parameters is you want to override the defaults:

1. `output.type`, as either `"darray"` or `"dframe"` (default is `"dlist"`).

2. `nparts`. Instead of just a scalar value, this parameter needs to be a vector of length 2, in order to specify how to two-dimensionally partition the output `darray` or `dframe`. For example, `nparts=c(2,2)` means you want the four partitions resulting from `dmapply` to be stitched together in a 2 by 2 fashion. We expect most people to use only single dimension such as row partitioned data, `nparts=c(N,1)`.
3. `combine` This argument is needed to fit the output of `dmapply` into the correct number of partitions. For example, there may be cases where `dmapply` returns 10 elements, but you have specified only 4 partitions in your output via `nparts`. In such as case, `combine` allows you to specify how elements should be combined to form only 4 partitions. You may like to think of this operation as what is called within each partition together on the results, to fit the output partitioning. `combine` can be either `c` (default), `rbind`, or `cbind`.

So, let's create a 4x4 `darray`, consisting of 4 2x2 partitions, where each partition contains values equal to its partition identifier:

```
my.darray2 <- dmapply(function(x) matrix(x,2,2), 1:4, output.type="darray", combine="rbind", nparts=c(2
my.darray2
```

```
##
## ddR Distributed Object
## Type: darray
## # of partitions: 4
## Partitions per dimension: 2x2
## Partition sizes: [2, 2], [2, 2], [2, 2], [2, 2]
## Dim: 4,4
## Backend: parallel
```

Even though we didn't `rbind` anything, as each iteration of `dmapply` was one partition of the result, the `combine` value was necessary. Since the default value of `combine` is `c`, which flattens and vectorizes the results within each partition (this is the default behavior of R's `mapply`). So `rbind` prevents this from happening, and the matrix structure is retained. We can look at what's stored in `my.darray2` by using the `collect` operator, which brings the data from the distributed backend to the local R instance, as a local R object:

```
my.array <- collect(my.darray2)
my.array
```

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

Collect() and parts()

As mentioned above, `collect` allows you to gather data from the partitions of a distributed object and convert it into a local R object.

You can gather individual partitions of the distributed object by using the second parameter of `collect`. For example, to get the third partition of our previous darray, `my.array2`, you can write:

```
collect(my.darray2,3)
```

```
##      [,1] [,2]
## [1,]    3    3
## [2,]    3    3
```

`parts` is a construct which takes a distributed object, and returns a `list` of new distributed objects, each of which represents one partition of the original distributed object. For example, let's take a look at `my.darray2` again:

```r
my.darray2
```

```
##
## ddR Distributed Object
## Type: darray
## # of partitions: 4
## Partitions per dimension: 2x2
## Partition sizes: [2, 2], [2, 2], [2, 2], [2, 2]
## Dim: 4,4
## Backend: parallel
```

Let's call `parts` on it:

```r
parts(my.darray2)
```

```
## [[1]]
##
## ddR Distributed Object
## Type: darray
## # of partitions: 1
## Partitions per dimension: 1x1
## Partition sizes: [2, 2]
## Dim: 2,2
## Backend: parallel
##
## [[2]]
##
## ddR Distributed Object
## Type: darray
## # of partitions: 1
## Partitions per dimension: 1x1
## Partition sizes: [2, 2]
## Dim: 2,2
## Backend: parallel
##
## [[3]]
##
## ddR Distributed Object
## Type: darray
## # of partitions: 1
## Partitions per dimension: 1x1
## Partition sizes: [2, 2]
## Dim: 2,2
## Backend: parallel
```

```
##
## [[4]]
##
## ddR Distributed Object
## Type: darray
## # of partitions: 1
## Partitions per dimension: 1x1
## Partition sizes: [2, 2]
## Dim: 2,2
## Backend: parallel
```

In the above example the output is a list of length 4, where each item is itself a `darray`, with partitioning and size equal to one partition of the original. We can also subset using `parts` to obtain just the second and third parts, respectively.

```r
parts(my.darray2,2:3)
```

```
## [[1]]
##
## ddR Distributed Object
## Type: darray
## # of partitions: 1
## Partitions per dimension: 1x1
## Partition sizes: [2, 2]
## Dim: 2,2
## Backend: parallel
##
## [[2]]
##
## ddR Distributed Object
## Type: darray
## # of partitions: 1
## Partitions per dimension: 1x1
## Partition sizes: [2, 2]
## Dim: 2,2
## Backend: parallel
```

The primary use of `parts` is to execute `dmapply` on partitions of the distributed objects. This is explained more in the next section.

Performing "work" with dmapply

When performing any computation with dmapply', the inputs to the function can be any combination of distributed objects (dlist,dframe,darray),partsof distributed objects, and standard R objects. More specifically,dlapplyanddmapply` statements generally take the following form:

```r
dlist1 <- dlapply(arg,FUN,nparts)
dlist2 <- dmapply(FUN,arg1,arg2,MoreArgs,nparts)
darray.or.dframe <- dmapply(FUN,arg1,arg2,MoreArgs,output.type,combine,nparts)
```

Valid types for the above arguments are the following:

1. FUN: any function with one or more arguments, defined using function in R.
2. arg*: any iterable collection

1) R objects: list, data.frame, matrix, any R vector, e.g., 1:10, or c(1,3,2)
2) distributed objects: dlist, dframe, and darray
3) parts of distributed objects dlist, dframe, and darray

3. MoreArgs: a list of (usually named) items that are also arguments to FUN, but are not iterated over, and instead passed to each iteration of the dmapply as a whole.
4. output.type: a string of either dlist, darray, dframe, or sparse_darray. By default, it is dlist.
5. combine: a string of either default, rbind, cbind, or c. The default default means c for darray and dframe, but nothing for dlist. For more information, please consult the user guide.
6. nparts: A numeric vector, of length 1 or 2. dlist objects can only have 1d-partitioning, but darray and dframe objects have 2d-partitioning.

When distributed objects are passed in dmapply, the semantics is same as R's lapply and mapply on regular R objects. This means FUN is applied per column in a dframe, once per item for dlist objects, and once per element (in column-major order) for darray objects.

When parts is used, a list of partitions of the underlying distributed object is returned. Therefore, dmapply operates on the list in the traditional manner, which means the function FUN is applied to each partition of the distributed object.

Scroll to end of this document for more examples on how to use dmapply on distributed objects and their partitions.

Operators

'ddR' supports a number of R-style, R-equivalent, operations on distributed objects. These are implemented "generically" based on dmapply, so they should work on all supported backends.

Examples:

```
## Head and tail
head(my.darray2,n=1)
```

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

```
tail(my.darray2,n=1)
```

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

```
## Subsetting
my.darray2[2,c(2,1)]
```

```
## [1] 1 1
```

```r
## Statistics
colSums(my.darray2)
```

```
## [1]  8  8 12 12
```

```r
max(my.darray2)
```

```
## [1] 4
```

There are many more!

Repartitioning

You may sometimes like to repartition your data. This can be done with the `repartition` command.

Say you have a 4x4 `darray` filled with 3:

```r
da <- darray(psize=c(2,2),dim=c(4,4),data=3)
da
```

```
##
## ddR Distributed Object
## Type: darray
## # of partitions: 4
## Partitions per dimension: 2x2
## Partition sizes: [2, 2], [2, 2], [2, 2], [2, 2]
## Dim: 4,4
## Backend: parallel
```

`da` is currently partitioned into 4 pieces of 2x2 arrays. You could repartition it to be two parts of 4x2 arrays. Currently this requires you to have another `skeleton` object against which to repartition your input. This object acts as the "model" by which your input should be repartitioned. The skeleton should have the same dimensions as the input, but a different partitioning scheme. For example:

```r
skel <- darray(psize=c(4,2),dim=c(4,4),data=0)
```

`skel`, like `da`, is also a 4x4 darray, but it is partitioned differently. In such cases `repartition(input,skeleton)` can be used to return a new distributed object that retains the data of `input`, but has the partitioning scheme of `skeleton`:

```r
da <- repartition(da,skel)
da
```

```
##
## ddR Distributed Object
## Type: darray
## # of partitions: 2
## Partitions per dimension: 1x2
## Partition sizes: [4, 2], [4, 2]
## Dim: 4,4
## Backend: parallel
```

As you can see, `da` is now partitioned just like `skel`. Executing `collect` shows that it still has the same data as before:

```
collect(da)
```

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

Note that `repartition` may be called implicitly and automatically by some backends during `dmapply` or `dlapply`. If the inputs and outputs (based on `nparts`) are not partitioned compatibly, each execution unit may not have the data required to process its chunk of computation. In this case, the backend may automatically call `repartition` on one or more of your inputs. In this case, performance may be impacted, so it is good practice to learn what results in compatible partitioning.

More Examples

Initializing a distributed list (`dlist`):

```
a <- dmapply(function(x) { x }, rep(3,5))
collect(a)
```

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

Printing `a`:

```
a
```

```
##
## ddR Distributed Object
## Type: dlist
## # of partitions: 5
## Partitions per dimension: 5x1
## Partition sizes: [1], [1], [1], [1], [1]
## Length: 5
## Backend: parallel
```

a is a distributed object in ddR. Note that we did not specify the number of partitions of the output, but by default it is equal to the length of the inputs (5). Use the parameter nparts to specify how the output should be partitioned:

Below is the code to add 1 to the first element of a, 2 to the second, etc. The syntax of dmapply is similar to R's standard mapply function.

```
b <- dmapply(function(x,y) { x + y }, a, 1:5,nparts=1)
b
```

```
##
## ddR Distributed Object
## Type: dlist
## # of partitions: 1
## Partitions per dimension: 1x1
## Partition sizes: [5]
## Length: 5
## Backend: parallel
```

As you can see, b only has one partition of 5 elements.

```
collect(b)
```

```
## [[1]]
## [1] 4
##
## [[2]]
## [1] 5
##
## [[3]]
## [1] 6
##
## [[4]]
## [1] 7
##
## [[5]]
## [1] 8
```

Some other operations: '

Adding a to b, then subtracting a constant value

```
addThenSubtract <- function(x,y,z) {
  x + y - z
}
c <- dmapply(addThenSubtract,a,b,MoreArgs=list(z=5))
collect(c)
```

```
## [[1]]
## [1] 2
##
## [[2]]
## [1] 3
```

```
##
## [[3]]
## [1] 4
##
## [[4]]
## [1] 5
##
## [[5]]
## [1] 6
```

Accessing dobjects by parts:

```r
d <- dmapply(function(x) length(x),parts(a))
collect(d)
```

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

We partitioned `a` with 5 parts and it had 5 elements, so the length of each partition is of course 1.

However, `b` only had one partition, so that one partition should be of length 5:

```r
e <- dmapply(function(x) length(x),parts(b))
collect(e)
```

```
## [[1]]
## [1] 5
```

For more example, check out our GitHub repo: https://github.com/vertica/ddR

Using the Distributed R backend

To use the Distributed R library for ddR, first install `distributedR` from https://github.com/vertica/DistributedR and `distributedR.ddR` from https://github.com/vertica/ddR.

Load the Distributed R driver library for ddR:

```r
library(distributedR.ddR)
```

```
## Loading required package: distributedR
## Loading required package: Rcpp
## Loading required package: RInside
## Loading required package: XML
## Loading required package: ddR
##
## Attaching package: 'ddR'
##
## The following objects are masked from 'package:distributedR':
##
##     darray, dframe, dlist, is.dlist
```

```r
useBackend(distributedR)
```

```
## Master address:port - 127.0.0.1:50000
```

Now you can try the different examples above which were used with the 'parallel' backend!