ddR User Guide

Edward Ma, Indrajit Roy

2015-10-20

ddR is both an API and an R package that permits the declaration of 'distributed' objects (i.e., `dlist`, `dframe`, `darray`), and facilitates 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 be selected for use with the API.

To get started, simply load the library and select a backend to use. If you'd like to use a custom backend that isn't `Parallel`, you'll also need to load the library or code containing the driver for that backend. (For example, if you'd like to use the driver packaged in distributedR.ddR, below, you'd also run the lines that are commented out)

```
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)
```

Creating Distributed Objects

There are two ways to create distributed objects in ddR.

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

Using the constructor functions

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

```
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 this printout 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 there are arugments.

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

```
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 that many 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 divided the data into lengths of 2, 2, and 1.

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

```
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's filled with 3s. Each partition is of size 2x2, and the dimensions of `my.darray` are 4x4. For `dframe`, the constructor follows the same format.

Using `dmapply` or `dlapply`

Constructors are handy when you want to initialize distributed objects quickly, but as `ddR` is a functional-programming API, every operation you do will generate new objects. This means that the primary way in which you will create distributed objects will be via `dmapply` and `dlapply`.

Here's how we would create the same `dlist` that we created above 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
```

What did I do? I ran the distributed `lapply` function, `dlapply`, on the function argument vector `1:5`, assigning to each element the value of the vector for that iteration. To specify `nparts`, I could also supply `nparts`, just as I did 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 API with `dmapply` for `darray` and `dframe` is slightly more complex, 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 need to also supply arguments for the following parameters:

1. `output.type`, as either `"darray"` or `"dframe"` (default is `"dlist"`).
2. `nparts`. Instead of just a scalar value, this 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.
3. `combine` This is needed if you ever have more than one `dmapply` iteration per partition. For example, you may `dmapply` on arguments of length 10, when you only have 4 partitions in your output `darray` or `dframe`. When that happens, `combine` allows you to specify how you'd like to combine this data. You may like to think of this operation as what is called within each partition together on the results, before the aggregate data of the partition assumes its form. `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 id. To do that, we can do the following:

```
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, parts

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

You may also pull an individual partition of the distributed object by using the second parameter of `collect`. For example, to get the third partition of our previous darray, `my.array2`, we can do:

```
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:

```
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
```

Now what happens when we run `parts` on it:

```
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
```

As you can see, we now have a list of length 4, each item is itself a `darray`, with partitioning and size equal to one partition of the original. We can also subset using `parts`:

```
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
```

to get just the second and third parts, respectively. The primary use of `parts` is to permit partition-based `dmapply`, where you would operate on one partition at a time. 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`,`dlapply`and`dmapply' statements generally take the following form:

```
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 `parts` is used, a list of partitions of the underlying distributed object is returned, so `dmapply` operates on that list in the traditional manner, which means you get to apply `FUN` to each partition of the distributed object. For all other types, the standard rules of R objects when used in `lapply`, `vapply`, and `mapply`

functions are followed: we apply by per column in a `data.frame`, once per item for `list` objects, and once per element (in column-major order) for `matrix` variables.

The distributed objects follow the same conventions as their regular R counterparts (per column of a `dframe`, per element of a `darray`, etc.).

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
```

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

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

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

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

There are many more, and many more to come!

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 3s:

```
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:

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

`skel`, like `da`, is also a 4x4 darray, but it's partitioned differently. Now we can do `repartition`. `repartition(input,skeleton)` returns a new distributed object that retains the data of `input`, but has the partitioning scheme of `skeleton`:

```
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 like how `skel` was. If we run `collect`, we see 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
```

We will soon add a way to easily (and cheaply) create skeleton objects (without having to initialize another distributed object that may be very large).

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

Init'ing a 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 now a distributed object in ddR. Note that we did not specify the number of partitions of the output, but by default it went to the length of the inputs (5). If we wanted to specify how the output should be partitioned, we can use the `nparts` parameter to `dmapply`:

Adding 1 to first element of `a`, 2 to the second, etc.

```
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:

```
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:

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

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

Note that `parts()` and non-parts arguments can be used in any combination to dmapply. `parts(dobj)` returns a list of the partitions of that dobject, which can be passed into dmapply like any other list. `parts(dobj,index)`, where `index` is a list, vector, or scalar, returns a specific partition or range of partitions of `dobj`.

We also have support for `darrays` and `dframes`. Their APIs are a bit more complex, and this guide will be updated shortly with that content.

For a more detailed example, you may view (and run) the example scripts under /examples.

Using the Distributed R backend

Use the Distributed R library for ddR:

```
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
```

```
useBackend(distributedR)
```

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

Now you can try the different list examples which were used with the 'parallel' backend.

How to Contribute

You can help us in different ways:

1. Reporting issues.
2. Contributing code and sending a Pull Request.

In order to contribute the code base of this project, you must agree to the Developer Certificate of Origin (DCO) 1.1 for this project under GPLv2+:

```
By making a contribution to this project, I certify that:

(a) The contribution was created in whole or in part by me and I have the
    right to submit it under the open source license indicated in the file; or
(b) The contribution is based upon previous work that, to the best of my
    knowledge, is covered under an appropriate open source license and I
    have the right under that license to submit that work with modifications,
    whether created in whole or in part by me, under the same open source
    license (unless I am permitted to submit under a different license),
    as indicated in the file; or
(c) The contribution was provided directly to me by some other person who
    certified (a), (b) or (c) and I have not modified it.
(d) I understand and agree that this project and the contribution are public and
    that a record of the contribution (including all personal information I submit
    with it, including my sign-off) is maintained indefinitely and may be
    redistributed consistent with this project or the open source license(s) involved.
```

To indicate acceptance of the DCO you need to add a `Signed-off-by` line to every commit. E.g.:

```
Signed-off-by: John Doe <john.doe@hisdomain.com>
```

To automatically add that line use the `-s` switch when running `git commit`:

```
$ git commit -s
```