Multiplyr basics

Jim Blundell

2016-05-15

Introduction

The multiplyr package is intended to provide simple and transparent parallel processing functionality through an interface similar to one already familiar to many R users (dplyr):

```
# Construct a new data frame
dat <- Multiplyr (x=1:100, G=rep(c("A", "B", "C", "D"), length.out=100))

# Group data by G
dat %>% group_by (G)

# Summarise length(x) in each group and store in N
dat %>% summarise (N = length(x))
```

To those unfamiliar with dplyr: this package makes heavy use of the %>% operator, which allows several operations to be chained together. dat %>% ... results in ... being applied to the data frame called dat. This allows multiple operations to be chained together. For example:

```
dat <- Multiplyr (x=1:100, y=1:100)

dat %>% filter(x<=50) %>% mutate (y=x*2) %>% select (y)
```

The details of what each of these functions are for is described elsewhere in this document.

Multiplyr is a reference class

There is one surprising thing missing from the example code here, which is the almost complete absence of <-, the assignment operator. Since these data frames represent shared memory, things like group_by modify it in place so assigning it back to the same object has no meaning. Attempting to assign the result of these to a different variable may have slightly unexpected behaviour. In the current version of this package only one data frame per cluster is supported.

Thinking in parallel

Expressions executed in parallel behave rather differently than you may be used to. Consider the following:

```
dat <- Multiplyr (x=1:100)

dat %>% mutate (x=length(x):1)
```

Reading this through, you would expect x to now contain the numbers from 100 to 1 in descending order. However, the answer depends entirely on how many cluster nodes that you have; each cluster will execute length(x):1 on its own subset of x and store the result in shared memory. If you have 2 cluster nodes then this will produce the numbers 50 to 1 repeated twice. If the data had been grouped, then this expression would be executed once for each group.

Text is not actually text

From the outside, Multiplyr data frames will appear to work like standard R data frames. However, the way text is stored internally is to convert them to factors, which is effectively:

1. Sort the data alphabetically
2. Find all the unique values and give them an ID number
3. Replace the data with that ID number
4. Store a lookup table mapping ID number to the actual text

This has one major advantage in that numeric data is easier and faster to manipulate and pass around. The major disadvantage, however, is that if a column has a lot of unique values of text then there's a large overhead in converting and passing to the cluster what these unique values are.

I wouldn't recommend using this package for data where you have columns with lots of unique values of text unless that text represents some sort of grouping.

Data order is not maintained

The internal representation of the data is a matrix and the way that the matrix is divided up amongst nodes is to give each node a contiguous block of that matrix (or several blocks in the case of grouped data). The side effect of this is that data must be sorted in a particular way. If data need to be in a particular order at the end, then use the arrange command at the end of your manipulations.

Creating a data frame

Creating a multiplyr data frame is achieved by calling Multiplyr and providing a comma separated list of column names and their values. You may also optionally specify a number of unnamed columns to allocate (e.g. for future calls to mutate) using alloc=N, where N is the number of columns. If you desire a specific number of nodes in the cluster be created then this may be specified with cl=N:

```
# Create a new data frame with 2 columns called x and y
dat <- Multiplyr (x=1:100, y=rep(1, 100))

# As above, but allocate space for 1 new column and create cluster with 2 nodes
dat <- Multiplyr (x=1:100, y=rep(1, 100), alloc=1, cl=2)

# Convert back to standard data frame
dat.df <- as.data.frame (dat)
dat.df <- dat %>% as.data.frame()
```

Manipulating

Create new columns with define

The first thing to note with creating new columns with a Multiplyr data frame is that the size of the underlying matrix is fixed at creation. This means that to create new columns you need to estimate in advance how many extra columns you might need. To allocate space for more columns on creation, use the alloc=N parameter of Multiplyr, where N is the number of extra columns to create. New columns are created with define, simply specifying their names and separating with commas.

```
# Create new data frame with space for 3 new columns
dat <- Multiplyr (x=1:100, alloc=3)

# Create 2 new columns named y and z
# Note that this define is actually not needed as mutate will define implicitly
dat %>% define (y, z)

# Do things with new columns
dat %>% mutate (y=x*2, z=sqrt(x))
```

As you will see in later sections, the above define operation was not actually technically necessary as mutate will create new columns if they do not already exist.

One time where define is not superfluous is when creating columns that will store factors or text. For this, along with the name of the new column, you need to specify a template to copy in the form of newcolumn=template:

```
# Create new Multiplyr frame with two columns (A and P) with space for 3 more
dat <- Multiplyr (A=rep(c("A", "B", "C", "D"), each=25),
                  P=rep(c("p", "q"), each=50),
                  alloc=3)

# Create new columns named newA and newP
dat %>% define (newA=A, newP=P)

# Set their values
dat %>% mutate (newA="A", newP="p")
```

Rename columns with rename

Renaming columns is easily achieved by specifying a list of newname=oldname pairs to rename:

```
# Create a new Multiplyr frame with variables named x, y and z
dat <- Multiplyr (x=1:100, y=100:1, z=rnorm(100))

# Rename to p, q and r
dat %>% rename (p=x, q=y, r=z)
```

Drop existing columns with undefine or select

There are two potential ways to drop columns from a data frame: specifying which columns to drop (undefine), or specifying which columns to keep (select):

```
# Create Multiplyr data frame with columns named x, y, z and misc
dat <- Multiplyr (x=1:100, y=100:1, z=rnorm(100), misc=rep(1, 100))

# Drop column named misc
dat %>% undefine (misc)

# Keep only the x and y columns
dat %>% select (x, y)
```

Grouping

Grouping data

One of the main strengths of the manipulations on data with multiplyr (as it is with dplyr) is that data may be arranged in groups first, with operations then taking place on individual groups.

Consider the experiment where 60 guinea pigs were given Vitamin C as either a supplement or in the form of orange juice at 3 different doses We can produce a table of summary statistics with the mean and standard deviation for each combination of supplement and dose as follows:

```
# Load data on guinea pig tooth growth:
#  len    Odontoblast (tooth cell) length
#  supp   Supplement (VC = Vit C supplement, OJ = Orange Juice)
#  dose   Dose in mg (0.5, 1 or 2)
#
data (ToothGrowth)

# Convert into a Multiplyr data frame with space for 2 new columns
dat <- Multiplyr (ToothGrowth, alloc=2)

# Group data by supplement and dose
dat %>% group_by (supp, dose)

# Produce summary statistics
dat %>% summarise (len.mean = mean(len), len.sd = sd(len))
```

Ungrouping and regrouping

Returning a data frame back to ungrouped form is achieved simply using ungroup. It is also possible to easily revert back to a grouped form using the same groupings with regroup.

```
# Load data on guinea pig tooth growth:
data (ToothGrowth)
dat <- Multiplyr (ToothGrowth)

# Group data by supplement and dose
dat %>% group_by (supp, dose)

# Ungroups data
dat %>% ungroup()

# Regroups data (implicitly by supplement and dose)
dat %>% regroup()
```

Manipulating rows

Filtering based on criteria with filter

A basic operation when working with data is to select a subset of rows based on certain criteria.

```
# Load data on guinea pig tooth growth:
data (ToothGrowth)
dat <- Multiplyr (ToothGrowth)

# Group by supplement
dat %>% group_by (supp)

# Select only guinea pigs with a dose >= 1 mg/kg
dat %>% filter (dose >= 1)

# Produce summary
dat %>% summarise (mean.len = mean(len))
```

Filtering out duplicates with distinct

Duplicate rows may be removed by calling distinct with no parameters. Alternatively, distinct may be used to find all unique combinations of a column or selection of columns:

```
# Create new Multiplyr data frame
dat <- Multiplyr (A=rep(c("A", "B", "C", "D"), each=25),
                  P=rep(c("p", "q"), length.out=100))

# Filter so only one of each AxP combination
dat %>% distinct()

# Filter so only one of each A
dat %>% distinct(A)
```

Note that if the data are grouped, then it will return one distinct entry per grouping.

Selecting a specific subset wih slice

Selecting a subset of data can be achieved using the slice function, by specifying the rows numerically, by range or as a logical vector:

```
# Construct a new data frame
dat <- Multiplyr (x=1:100, G=rep(c("A", "B", "C", "D"), length.out=100))

# Return the first 10 rows only
dat %>% slice (1:10)

# Return alternate rows
dat %>% slice (rep(c(TRUE, FALSE), length.out=10))
```

By default slice will return a subset of those rows treating the data frame as a single contiguous entity. This does not always make sense when data have been grouped, so slice may also be used to obtain a subset of data in each group by specifying each=TRUE:

```
# Construct a new data frame
dat <- Multiplyr (x=1:100, G=rep(c("A", "B", "C", "D"), length.out=100))
```

```
# Group by G
dat %>% group_by (G)

# Return the first 10 rows in each group
dat %>% slice (1:10, each=TRUE)
```

Similarly, each=TRUE may be used to obtain a subset of data in each node where grouping has not been done:

```
# Construct a new data frame
dat <- Multiplyr (x=1:100, G=rep(c("A", "B", "C", "D"), length.out=100))

# Return the first 10 rows in each node
dat %>% slice (1:10, each=TRUE)
```

Sorting data with arrange

Sorting data within columns is achieved with the arrange function, specifying the columns to sort by separated by commas (the extra columns are used to determine how to break ties). For example:

```
# Create new data frame
dat <- Multiplyr (G=rep(c("A", "B", "C", "D"), each=25), x=100:1)

# Sort by G, then by x
dat %>% arrange (G, x)
```

Manipulating data

Updating values with mutate

Modification of the actual data stored within a data frame is done through the use of mutate. Multiple mutation operations may be separated with a comma and are given in the form of colname=expression. If the column name specified does not exist, then mutate will attempt to implicitly define it:

```
# Construct a new data frame with space for 2 new columns
dat <- Multiplyr (x=1:100, alloc=2)

# Update all cells in the x column to be twice their value
dat %>% mutate (x=x*2)

# Create 2 new columns and populate them with data
dat %>% mutate (y=x*2, z=sqrt(x))
```

Updating values and dropping other columns with transmute

The transmute function works in very much the same way as mutate, but drops any columns not explicitly specified. It's effectively mutate, followed by select.

```
# Construct a new data frame with space for 2 new columns
dat <- Multiplyr (x=1:100, alloc=2)

# Create new columns (y and z), drop x in the process
dat %>% transmute (y=x*2, z=sqrt(x))
```

Summarising data with summarise and reduce

Summarising data is achieved using the summarise command, with each expression returning a single value. We could produce a summary table of mean and standard deviation of guinea pig tooth lengthdata grouped by supplement and dose with the following:

```
# Load data on guinea pig tooth growth:
data (ToothGrowth)

# Convert into a Multiplyr data frame with space for 2 new columns
dat <- Multiplyr (ToothGrowth, alloc=2)

# Group data by supplement and dose
dat %>% group_by (supp, dose)

# Produce summary statistics
dat %>% summarise (len.mean = mean(len), len.sd = sd(len))
```

The results for when data are not grouped are slightly less straightforward. Consider the following:

```
dat <- Multiplyr (x=1:100)

dat %>% summarise (N=sum(x))
```

This will not produce a single result: it will produce one result for each cluster. Therefore, a second summarise has to be done, but in such a way that it is guaranteed to be executed once. This is done using the reduce function:

```
dat <- Multiplyr (x=1:100)

dat %>% summarise (N=sum(x)) %>% reduce(N=sum(x))
```

Executing arbitrary code using within_node and within_group

Two functions are provided for making parallel processing more convenient. The first of these is within_node, which executes a block of code once for each node in a cluster. It acts as a persistent environment, so that subsequent calls to within_node or other operations, such as summarise may be make use of variables created within that environment:

```
# Construct a new data frame
dat <- Multiplyr (x=1:100)

# Define y and z to be new columns
dat %>% define(y, z)
```

```
# Execute the following code within each node
dat %>% within_node ({
    y <- x * 10
    z <- sqrt(y)
})
```

Similarly, code may be executed within each group using within_group:

```
# Construct a new data frame
dat <- Multiplyr (x=1:100, G=rep(c("A", "B", "C", "D"), length.out=100))

# Group by G
dat %>% group_by (G)

# Execute the following block of code within each group
dat %>% within_group({
    N <- length(x)
    xbar <- sum(x) / N
})

# Export the data
dat %>% summarise(N=N, xbar=xbar)
```

The necessity of exporting the data using summarise rather than define may not be immediately obvious, now why does multiplyr not simply create N and xbar as new columns by default in the within_node or within_group code. The role of within_group and within_node is to provide an environment for executing code where variables are bound to the columns to allow for more complex operations, e.g. running models. This means that there is no restriction on the type of data that may be stored, i.e. model objects. For example:

```
# Create data frame
dat <- Multiplyr (G = rep(c("A", "B"), each=50),
                  m = rep(c(5, 10), each=50),
                  alloc=1)

# Group by G
dat %>% group_by (G)

# Generate some random data in x with mean of m
dat %>% mutate (x=rnorm(length(m), mean=m))

# Fit a linear model with just an intercept to x in each group
dat %>% within_group ({
    mdl <- lm (x ~ 1)
})

# Extract intercept and store it in x.mean
dat %>% summarise (x.mean = coef(mdl)[[1]])
```

No strings attached

Data within a Multiplyr data frame is internally represented numerically, regardless of whether it's actually numeric or not. At first this may seem like a limitation, but can potentially make data manipulation very,

very fast. When data in a Multiplyr data frame is updated, e.g. setting a value to "A", this results in "A" being looked up for that column's factor levels and then assigned to the underlying numeric matrix.

However, there are times that the actual, specific content of the column is not relevant. One way to speed up certain operations is therefore to put the data frame into "no strings attached" mode, which disables any of these look-ups. For example:

```
# Construct a new data frame
dat <- Multiplyr (G=rep(c("A", "B", "C", "D"), length.out=100),
                  H=rep(c("p", "q", "r", "s"), each=25))
# Display data
dat["G"]
dat["H"]

# Switch into NSA mode
dat %>% nsa()
dat["G"]
dat["H"]

# Do some things
dat %>% mutate (G=max(G))

# Switch back
dat %>% nsa(FALSE)
dat["G"]
dat["H"]
```

Speed considerations/limitations

There are some things that are particularly slow with a Multiplyr data frame. In particular, the way in which character data is handled. Any character data is effectively converted into a factor transparently. This means that if there is a column of text where every cell is unique, then this is passed on in its entirety, which is very slow.

The initial creation of a parallel data frame is slow: the reason for this is that in the background a new local cluster is started. If you have 4 cores or CPUs then 4 new instances of R will be started, which can take a few seconds.

This package is not (yet!) a complete replacement for dplyr and there are some notable things missing, most notably neat interfacing with SQL databases. Currently only one parallel data frame per cluster is supported, so there also are't any join operations.