Some remarks on chunked lopping in packages bit, ff, and R.ff

Jens Oehlschlägel

March 15, 2010

Abstract

Chunked looping plays a core role in working with large objects. It helps reducing RAM requirements and can speed up calculation by parallelization. This document discusses older and newer options to loop along one or more large `ff` or `bit` vectors and do someting meaningful with them.

Packages `bit`, `ff` and `R.ff` extend `R`'s capabilities to handle large datasets. While the released packages `bit` and `ff` focus on basic infrastructue for creating and accessing large objects, a future package `R.ff` focuses on processing with large objects. Currently `R`.ff is just an experimental stub, the final version may differ greatly. One reason is the multitude of parallel processing options for R – of which none has emerged as a clear standard so far.

Contents

1 Chunked looping with ffapply and friends

Let's create a simple example of a ffdf dataframe with 4 columns

```
> require(R.ff)
> n <- 1e8
> a <- ff(levels=letters, vmode="byte", length=n)
> b <- ff(levels=LETTERS, vmode="byte", length=n)
> x <- ff(vmode="double", length=n)
> y <- ff(vmode="double", length=n)
> d <- ffdf(a,b,x,y)
```

Assume we want to fill vector x with random numbers. In standard R we could simply write:

```
> x[] <- rnorm(length(x))
```

But we cannot call `rnorm()` for creating a very large vector. Instead we must loop over chunks of the vector and fill each chunk with a call of `rnorm()` such that each call fits into RAM. Since ff version 2.0 released August 2008 we have ffvecapply and friends to simplifiy this:

```
> ffvecapply( x[i1:i2] <- rnorm(i2-i1+1), X=x)
> x

ff (open) double length=100000000 (100000000)
        [1]         [2]         [3]         [4]         [5]         [6]
-0.09949083  0.29794782  0.13795183 -1.63020215  1.26881346  1.05774159
        [7]         [8]              [99999993]  [99999994]  [99999995]
-1.19560657  0.69664356           :  1.11075088  0.05737748 -0.35772872
 [99999996]  [99999997]  [99999998]  [99999999] [100000000]
 0.47335014  0.84945972  0.85297214  2.41942421  3.12373692
```

It will automatically loop over chunks of positions i1:i2 and supports a variety of convenience functions like automatic creation of return values. However, the ffapply functions have been marked as preliminary, for example do not support parallelism and are a bit unusual in supporting expressions rather than functions.

2 Implicit chunked looping with S3-methods

At UseR!2008 we had presented a protoype of R.ff, a packagce aiming at turning R into a system that behaves like R but seemlessly supports large objects. In this unpublished package we compiled many standard R functions to corresponding S3 methods, which had allowed us to simply write

```
> z <- x + y
```

to obtain a new `ff` vector `z` as the sum of `ff x` and `ff y`. However the problem with such approach is that a more complex expression like

```
> z2 <- x + y + z
```

will not only write one result vector. Instead it will also write intermediate results (here x+y) to disk which is inefficient. Therefore we experimented with wrapping more complex expressions into a specific parser/evaluator as in

```
> ffbatch( z2 <- x + y + z )
```

which gives nice performance improvements over the method dispatch approach. However, one learning remains true: an attempt to create a system in which the user/programmer does not need to be aware of the fact that he uses special (`ff`) objects will either not reach the flexibility of R or it will come at huge performance penalties. For example in `ff` we have made the design choice to return a standard R ram object when subscripting an `ff` object. Returning instead the identical class (`ff`) would reduce the need to treat `ff` objects differently from standard R objects, however returning `ff` could easily kill performance: by writing a new `ff` to disk for each subscripting operation. Well, unless `ff` objects would support *virtual* subscripting, i.e. return an object decorated with the subscript information but referring to the same file.

3 Virtual subscripting

In `ff` we have experimented with virtualizaton: `ff` has a functionality called *virtual windows* (`vw`). An `ff` vector or array can pretend to be a smaller subset of its data, but only one contiguous selection in each dimension. This allows very interesting ways to work with `ff` objects: we can `split` an `ff` object virtually into smaller *cubelets* and process these with some standard `apply` function, potentially on multiple cores or cluster nodes. Repeating an exampe from our presentation on `R.ff`, we would create a big array

```
> Cube <- ff(vmode="double", dim=c(1000,1000,1000))
```

then split into cubelets

```
> Cubelets <- fftile(Cube, ntile=c(10,10,10))  # only 1 sec
```

and finally apply `someFUN` to each cublet

```
> apply(Cubelets, 1:3, someFUN)
```

Although an `ff` object with a vw attribute will behave like a smaller object, subscripting from it will return a standard ram object, not an `ff` object. The implementation of the vw was already complicated – given complications like `dimorder`. It would be very challenging to generalize the virtualization to a point where *any* subscript selection can be represented in a virtual way such that we return a virtually smaller object from subscripting. This would mean that much of the subscript processing would happen on virtual attributes – not touching the data at all. But what if the number of selected elements is large, say too large to fit into ram? The ram-saving rle-representation that we use in hybrid indexes (`hi`) are not very suitable for recursive subset operations. Beyond that, accessing a chunk of data would be very indirect: first operate on virtual selection information, then retrieve the data to ram. Similarly redundant overhead is associated with the cubelet approach: each cubelet replicates a lot of information beyond the pure subscript information. These considerations lead us in summary to focus on new more efficient subscript types: those published in package `bit`.

4 Explicit chunking with range indices

Package `bit` was released in October 2009 together with `ff` version 2.1. It comes with several innovative subscript types for selecting from vectors. Given the fact that `ff` objects are currently limited to a length of about 2 billion elements and that modern computers have several GB of RAM, the `bit` type is a powerful candidate for efficient in-ram representations of selections and fast operations on those. Many operations on `bit` vectors support chunked access using another new subscript type: range indices `ri` simply represent a contiguous chunk of positions. They carry a start posistion, a stop position and optionally the total length of the subscripted object. The generic function `chunk` returns a list with such `ri` chunk definitions.

```
> chunks <- chunk(x)
```

look at the first two chunks

```
> chunks[1:2]

[[1]]
range index (ri) from 1 to 2000000 maxindex 100000000

[[2]]
range index (ri) from 2000001 to 4000000 maxindex 100000000
```

Looping over chunks is as easy as this

```
> for (ch in chunks)
+   y[ch] <- rnorm(sum(ch))
```

where `sum(ch)` returns the number of selected elements in the chunk (think this as if `ch` where a logical selection vector like `logical`, `bit` or `bitwhich`). This representation of all chunks in a loop by a list with `ri` objects is very light-weight and thus compatible with the many instances of `lapply` in `R`, namely functions that support parallelization as in snowfall:

5 Semi-explicit chunked looping

The above loops with `for` or `snowfall` can be simplified with function `ffchunk`, which has an interface marrying features from `ffvecapply` and `chunk`. We can submit an expression with explicit mention of loop indices, the chunking and runnning the loop happends implicitely – but can be customized. The above example becomes

```
> ffchunk( y[i] <- rnorm(sum(i)) )

Library ff loaded.
```

If `R` has been started with snowfall suppport (e.g. `Rgui.exe -parallel -cpus=2`) this will be executed on multiple snowfall slaves, otherwise it runs locally in a serial loop. If snowfall is not initialized, `ffchunk` will `sfInit` and `sfStop` on exit, however it is recommended to initialized and stop snowfall outside of `ffchunk` in order to avoid the associated overhead with each call of `ffchunk`. `ffchunk` will automatically load `bit` and `ff` and export all objects to all snowfall slaves. `ffchunk` will try to guess a good chunk size (which the user can overwrite using arguments `chunks=, from=, to=, by=, length.out=, RECORDBYTES=, BATCHBYTES=`. `ffchunk` will try to guess whether the expression should return or not such that we get a return value from writing an expression without assignment, as in

```
> ffchunk( summary(x[i]) )[1:2]

Library ff loaded.
[[1]]
     Min.   1st Qu.    Median      Mean   3rd Qu.      Max. 
-4.783000 -0.675500 -0.002237 -0.001495  0.672500  4.892000 

[[2]]
      Min.    1st Qu.     Median       Mean    3rd Qu.       Max. 
-4.7170000 -0.6752000 -0.0001951 -0.0006000  0.6745000  5.0300000 
```

or from a multiple expression as in

```
> ret<- ffchunk({
+   s <- sample(letters, sum(i), TRUE)
+   a[i] <- s
+   na <- sum(s=="a")
+   s <- sample(LETTERS, sum(i), TRUE)
+   b[i] <- s
+   nA <- sum(s=="A")
+   return(c(na, nA))
+ }, VERBOSE=TRUE)
```

```
Library ff loaded.
TOTAL time= 38.29 sec
```

6 Chunked bit-filtering

We mentioned that `bit` operations support chunking and give a few examples here.

First we create two `ff` boolean vectors which we fill by evaluating logical conditions a parallel loop, once done we coerce to `bit`

```
> system.time({
+   bool1 <- ff(vmode="boolean", length=n)
+   bool2 <- ff(vmode="boolean", length=n)
+   ffchunk({ bool1[i] <- a[i]=="a"; bool2[i] <- b[i]=="A" })
+   both <- as.bit(bool1) & as.bit(bool2)
+   both
+ })
```

```
Library ff loaded.
   user  system elapsed
   0.59    0.20   27.56
```

In the next call we directly fill a local `bit` vector, since this is not a very small object, we avoid sending it to snowfall slaves and ecxecute locally in a serial loop

```
> system.time({
+   bit1 <- bit(n)
+   bit2 <- bit(n)
+   ffchunk({ bit1[i] <- a[i]=="a"; bit2[i] <- b[i]=="A" }, parallel=FALSE, VERBOSE=TRUE)
+   both <- bit1 & bit2
+ })
```

```
TOTAL time= 29.84 sec
   user  system elapsed
  26.73    3.16   29.96
```

If the number of selected elements is low as in

```
> sum(both)
```

```
[1] 147982
```

then we can directly use the `bit` filter on an `ff` object as in

```
> sum(d$x[both])
```

```
[1] 1000.586
```

If the number of selected elements is high as in

```
> sum(!bit1)

[1] 96154732
```

then we need chunked looping, either directly using a `bit` vector

```
> sum(unlist( ffchunk( sum( x[i][bit1[i]] ) , parallel=FALSE, VERBOSE=TRUE) ))

TOTAL time= 6.89 sec
[1] -3570.86
```

or indirectly using an `ff` boolean vector and benefitting from parallelization

```
> sum(unlist( ffchunk( sum( x[i][bool1[i]] ) , VERBOSE=TRUE) ))

Library ff loaded.
TOTAL time= 5.69 sec
[1] -3570.86
```

The latter can be tuned by taking out a redundant coercion from `ri` to `hi`

```
> sum(unlist( ffchunk({ h <- as.hi(i); sum( x[i][bool1[i]] ) }, VERBOSE=TRUE) ))

Library ff loaded.
TOTAL time= 5.73 sec
[1] 0
```

Note that both solutions have the disadvantage that the complete vector `x` needs to be read from disk BEFORE filtering it. More efficient is filtering first, i.e. combine the `ri` of each chunk with the `bit` filter

```
> sum(unlist( ffchunk( sum( x[as.which(bit1, range=i)] ) , parallel=FALSE, VERBOSE=TRUE) ))

TOTAL time= 1.81 sec
[1] -3570.86
```

or using the an `ff` boolean - which requires explicitely calling return():

```
> sum(unlist( ffchunk({w <- (i[[1]]:i[[2]])[bool1[i]]; return(sum( x[w] )) }, VERBOSE=TRUE) ))

Library ff loaded.
TOTAL time= 3.75 sec
[1] -3570.86
```

Note that the follwing is slower

```
> sum(unlist( ffchunk( sum( x[(i[[1]]:i[[2]])[bool1[i]]] ) , VERBOSE=TRUE) ))

Library ff loaded.
TOTAL time= 7.09 sec
[1] -3570.86
```

You might have wondered, why it was necessary to write

```
> as.which(bit1, range=i)
```

instead of simply

```
> i & bit1
```

Due to a strange design-decision in R's S3 class system, we cannot write methods that reliably dispatch on two user-defined classes. If we have two methods `&.bit` and `&.ri` the following expression

```
> riobj & bitobj
```

will neither dispatch on `&.bit` nor on `&.ri` but instead dispatch on an unsuitable method and report:

```
    Warning message:
Incompatible methods ("&.bit", "&.ri") for "&"
```

If R would in case of conflicting classes simply dispatch on the first argument, we could take control and define appropriate methods that resolve the class conflicts.

```
> "&.ri" <- function(e1, e2){
+   switch(class(e2)
+   , "bit" = as.bitwhich(e2, range=e1)
+   , as.bitwhich(as.bit(e1) & as.bit(e2))
+   )
+ }
> "&.bit" <- function(e1, e2){
+   switch(class(e2)
+   , "ri" = as.bitwhich(e2, range=e1)
+   , e1 & as.bit(e2)
+   )
+ }
```

However, the current behaviour of R does not allow to take control.

7 Wrap-up

Finally, let's not forget to stop snowfall.

```
> sfStop()
```