DISCLOSED

A first glimpse into 'R.ff'

(a package that virtually removes R's memory limit)

Oehlschlägel, Adler, Nenadic, Zucchini

Munich, Göttingen

August 2008

SUMMARY

The availability of large atomic objects through package 'ff' can be used to create packages implementing statistical methods specifically addressing large data sets (like subbagging or package biglm). However, wouldn't it be great if we could apply all of R's functionality to large atomic data? Package 'R.ff' is an experiment to provide as much as possible of R's basic functionality as 'ff-methods'. We report first experiences with porting standard R functions to versions operating on ff objects and we discuss implications for package authors (and maybe also R core). Instead of a summary, here we just quicken your appetite through the list of functions and operators where we have first experimental ports:

! != %% %*% %/% & | * + - / < <= == > >= ^ abs acos acosh asin asinh atan atanh besselI besselJ besselK besselY beta ceiling choose colMeans colSums cos cosh crossprod cummax cummin cumprod cumsum dbeta dbinom dcauchy dchisq dexp df dgamma dgeom dhyper digamma dlnorm dlogis dnbinom dnorm dpois dsignrank dt dunif dweibull dwilcox exp expm1 factorial fivenum floor gamma gammaCody IQR is.na is.nan jitter lbeta lchoose lfactorial lgamma log log10 log1p log2 logb mad order pbeta pbinom pcauchy pchisq pexp pf pgamma pgeom phyper plnorm plogis pnbinom pnorm ppois psigamma psignrank pt punif pweibull pwilcox qbeta qbinom qcauchy qchisq qexp qf qgamma qgeom qhyper qlnorm qlogis qnbinom qnorm qpois qsignrank qt quantile qunif qweibull qwilcox range range rbeta rbinom rcauchy rchisq rexp rf rgamma rgeom rhyper rlnorm rlogis rnbinom rnorm round rowMeans rowSums rpois rsignrank rt runif rweibull rwilcox sample sd sign signif sin sinh sort sqrt summary t tabulate tan tanh trigamma trunc var

R.ff DESIGN GOALS: THE WORDS LARGEST 'POCKET CALCULATOR'

large data

- being able to process large objects (size>RAM)
- many objects (sum(sizes)>RAM)

as convenient as possible

- R typical handling
- ff method dispatch
- transparent tempfile handling

as compatible as possible

- avoid duplicate implementation
- re-use existing functions

maximum performance

- close to in-RAM performance if size<RAM
- still able to process if size>RAM
- avoid redundant access
- allow tempfile re-use (because in some fs creating files is costly)

Source: Oehlschlägel, Adler, Nenadic, Zucchini (2008) A first glimpse into 'R.ff'

STANDARD PARAMETERS IN MANY FF FUNCTIONS

```
# would be nice
<-.ff
# but has too many complications, instead:

ff(...
, FF_RETURN  = TRUE    # bi-boolean in constructor: TRUE or FALSE
, BATCHSIZE  = NULL    # default .Machine$integer.max
, BATCHBYTES = NULL    # default (mostly) 1*getOption("ffbatchbytes")
, VERBOSE    = FALSE
)

ffvecapply(...
, FF_RETURN  = TRUE    # tri-boolean otherwise: TRUE or FALSE or
                       # or pass-in the return ff object
, BATCHSIZE  = NULL
, BATCHBYTES = NULL
, VERBOSE    = FALSE
)
```

Source: Oehlschlägel, Adler, Nenadic, Zucchini (2008) A first glimpse into 'R.ff'

FACILITATED CHUNKED LOOPING IN FF

PRELIMINARY

ffvecapply, ffrowapply, ffcolapply, ffapply

```
library(ff)

x <- ff(vmode="double", length=1e7)
ffvecapply( x[i1:i2] <- runif(i2-i1+1) + runif(i2-i1+1)
, X = x
, BATCHSIZE = 1e6
, VERBOSE = TRUE
)
x
```

Source: Oehlschlägel, Adler, Nenadic, Zucchini (2008) A first glimpse into 'R.ff'

A SHORT R.ff DEMO

```
library(R.ff)
bigR()
options(ffbatchbytes=2^22)
options(ffpagesize=2^20)
options(ffcaching="mmnoflush") # "mmeachflush"

system.time( x <- runif.ff(1e7) + runif.ff(1e7) )
print(x, maxlength=4)
memory.size(max=FALSE)  #  27 MB
memory.size(max=TRUE)   #  31 MB

system.time( x <- runif(1e7) + runif(1e7) )
memory.size(max=FALSE)  # 240 MB
memory.size(max=TRUE)   # 242 MB

# 6.6 sec R.ff mmeachflush
# 3.0 sec R.ff mmnoflush
# 2.7 sec   ff mmeachflush
# 1.7 sec   ff mmnoflush
# 1.5 sec    R pure RAM
```

Source: Oehlschlägel, Adler, Nenadic, Zucchini (2008) A first glimpse into 'R.ff'

COERCION TO FF FUNCTION ...

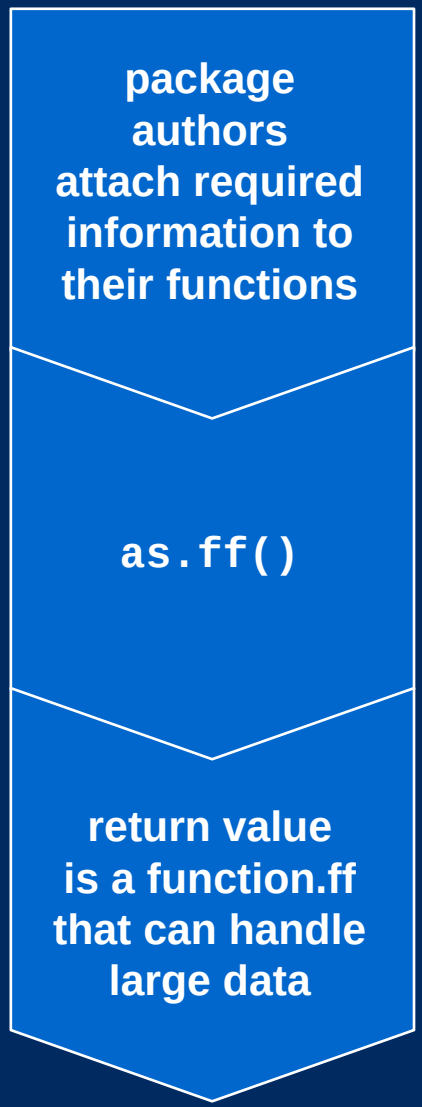

```
runif.ff <- as.ff(runif)
```

```
> runif.ff
function (n, min = 0, max = 1
, FF_RETURN = TRUE, BATCHSIZE = .Machine$integer.max
, BATCHBYTES = getOption("ffbatchbytes"), VERBOSE = FALSE)
{
    FF_ATTR <- list(vmode = "double", length = as.integer(n))
    FF_RET <- ffreturn(FF_RETURN = FF_RETURN, FF_PROTO = NULL
    , FF_ATTR = FF_ATTR)
    ffvecapply(
      EXPR = FF_RET[FF_I1:FF_I2] <- runif(FF_I2 - FF_I1 + 1L
                                    , min = min, max = max)
    , N = n, VMODE = "double"
    , FROM = "FF_I1", TO = "FF_I2", BATCHSIZE = BATCHSIZE
    , BATCHBYTES = BATCHBYTES, VERBOSE = VERBOSE
    )
    FF_RET
}
```

Source: Oehlschlägel, Adler, Nenadic, Zucchini (2008) A first glimpse into 'R.ff'

… HOW `as.ff.function` WORKS CONCEPTUALLY …

PRELIMINARY



- data types of arguments
- which arguments to recycle
- type of required processing (elementwise, aggregating, …)
- data type and structure of return value

- calling as.ff
- computing on the language

- recycles arguments automatically
- creates ff return object automatically
- can be customized using standard arguments
  - `FF_RETURN = TRUE`
  - `BATCHSIZE = .Machine$integer.max`
  - `BATCHBYTES = getOption("ffbatchbytes")`
  - `VERBOSE = FALSE`
- `method dispatch may be used to call function.ff`

Source: Oehlschlägel, Adler, Nenadic, Zucchini (2008) A first glimpse into 'R.ff'

… AND HOW as.ff.function WORKS PRACTICALLY

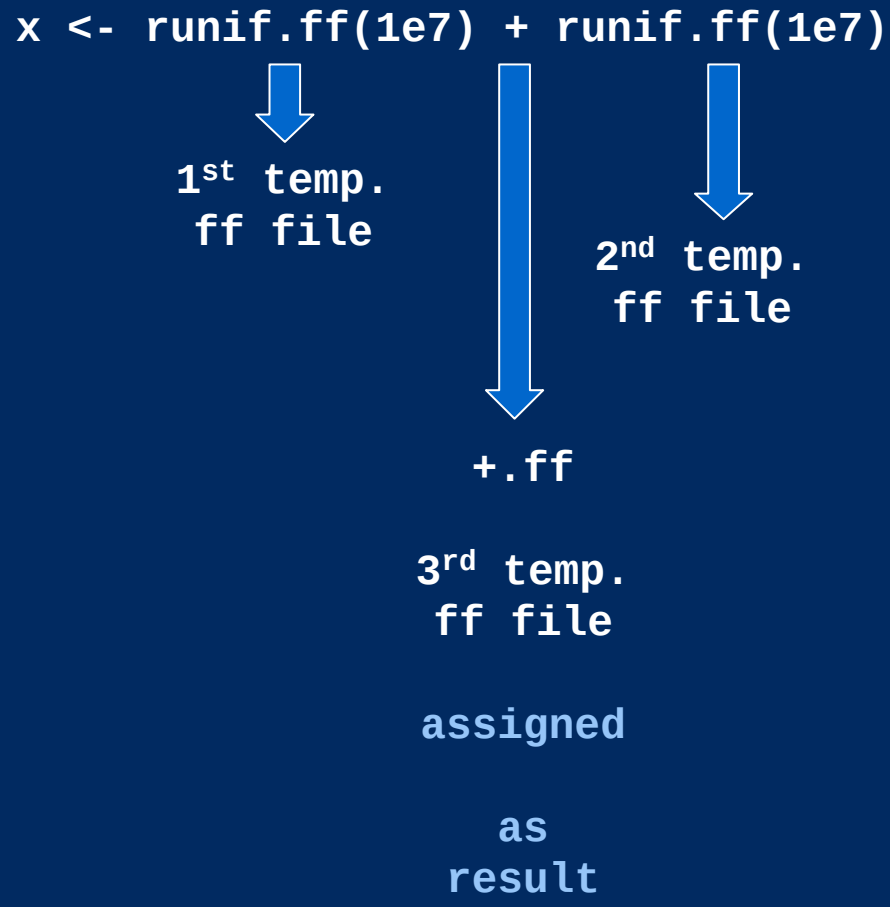
PRELIMINARY

```
funmode(runif) <- "fun1one2many"  # now inherits(runif, "fun1one2many")
funmeta(runif) <- list(vmode="double")     # attach further information
> as.ff.fun1one2many
function (x, vmode = "guess", ...){
    if (is.character(x)) { xid <- as.symbol(x); xfun <- get(x)
    }else{ xid <- substitute(x); xfun <- x }
    if (is.null(vmode)) stop("vmode required") else if (vmode == "guess"){
      fm <- funmeta(xfun)
      if (is.na(match("vmode", names(fm)))) {
         stop("vmode neither as argument nor as funmeta nor have we guessing")
      }else{ vmode <- fm$vmode }}
    xargs <- alistformals(xfun)
    yargs <- alist(FF_RETURN = TRUE, BATCHSIZE = .Machine$integer.max,
        BATCHBYTES = getOption("ffbatchbytes"), VERBOSE = FALSE)
    yvars <- c("FF_N", "FF_RET", "FF_ATTR", "FF_I1", "FF_I2")
    if (!all(is.na(match(names(xargs), c(names(yargs), yvars)))))
        stop("argument name conflict")
    ffargs <- c(xargs, yargs); callargs <- xargs
    for (i in names(xargs)) callargs[[i]] <- as.name(i)
    names(callargs)[1] <- ""; arg1nam <- as.name(names(xargs)[1])
    callargs[[1]] <- substitute(FF_I2 - FF_I1 + 1L, list(x = arg1nam))
    xcall <- as.call(c(list(xid), callargs))
    ffbody <- substitute({ FF_ATTR <- list(vmode = vmodeval_, length = as.integer(x))
        FF_RET <- ffreturn(FF_RETURN = FF_RETURN, FF_PROTO = NULL, FF_ATTR = FF_ATTR)
        ffvecapply(EXPR = FF_RET[FF_I1:FF_I2] <- xcall, N = x, VMODE = vmodeval_
        , FROM = "FF_I1", TO = "FF_I2"
        , BATCHSIZE = BATCHSIZE, BATCHBYTES = BATCHBYTES, VERBOSE = VERBOSE)
        FF_RET
    }, list(xcall = xcall, x = arg1nam, vmodeval_ = vmode))
    fffun <- function(){}; formals(fffun) <- ffargs; body(fffun) <- ffbody
    return(fffun)}
```

Source: Oehlschlägel, Adler, Nenadic, Zucchini (2008) A first glimpse into 'R.ff'

SEPERATELY DISPATCHED METHODS HAVE PERFORMANCE LIMITS

too many temporary files



A BATCH EVALUATOR FOR ELEMENTWISE FF EXPRESSIONS

ffbatch()

```
x <- ff(1:10000000, vmode="double")
y <- ff(1:10000000, vmode="double")
z <- ff(1:1000000, vmode="double")

# using separate ff method dispatch
a <- x + x^2 * 2 + x^3 * 3 + pi + y + z
# 25 .. 29 sec == 100%

# evaluating the complete expression in batches
a <- ffsimplebatch( x + x^2 * 2 + x^3 * 3 + pi + y + z )
```

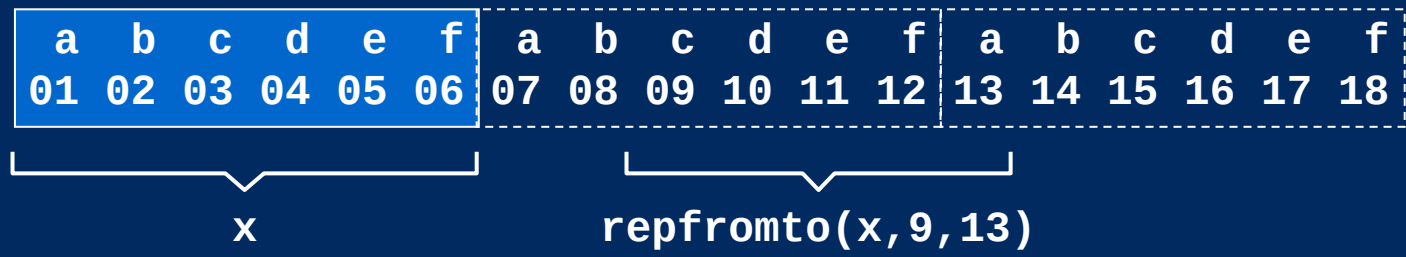
```
# ffvecapply( repfromto(x, i1, i2) + repfromto(x, i1, i2)^2 * 2 + …
# 8.6 .. 9.9 sec == 30% .. 40%

# save multiple reading of x and unnecessary repfromto()
a <- ffbatch( { b <- x ; b + b^2 * 2 + b^3 * 3 + pi + y + z } )
# 4.7 .. 5.9 sec == 16% .. 24%

# R RAM: 2 sec == 7% .. 8%
```

Source: Oehlschlägel, Adler, Nenadic, Zucchini (2008) A first glimpse into 'R.ff'

CHUNKED RECYCLING OF ARGUMENTS

repfromto(x, from, to)

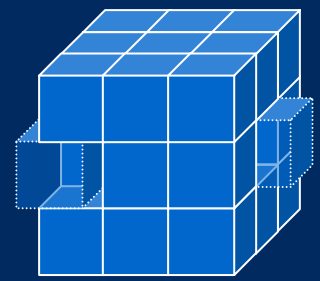
```
x <- ffvecapply( runif(i2-i1+1, max=repfromto(letters[1:6], i1, i2) )
, VMODE = "double"
, N = 1e7
, RETURN = TRUE
)
x
```



VIRTUAL CUBELETS ...

fftile()

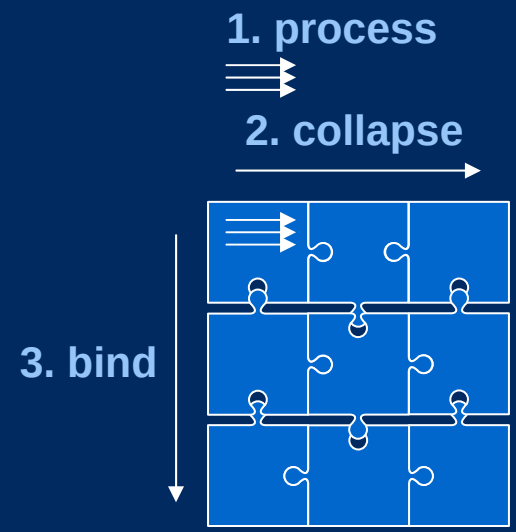
```
# Cube with 1000x1000x1000 cells
Cube <- ff(vmode="double", dim=c(1000,1000,1000)
, pagesize=2^20, caching="mmeachflush")

# split into cublets with 100x100x100 each,
# arranged in a metacube with 10x10x10 cublets
Cubelets <- fftile(Cube, ntile=c(10,10,10))  # only 1 sec
Cubelets
Cubelets[[2,3,4]]
# loop over Cubelets
apply(Cubelets, 1:3, function(l)print(l[[1]]))

# same loop over Cubelets but having access to indices and dimnames
indices <- fftile(Cube, c(10,10,10), what="indices")
apply(indices, 1:3, function(i){
  undropped <- do.call("["
  , c(list(Cubelets), as.list(i[[1]]), list(drop=FALSE)))
  print(undropped[[1]])
  }
)
```



VIRTUAL CUBELETS ... FOR EXAMPLE TILED rowSums()

fftile()

```
# Matrix with 10000x10000 cells
Matrix <- ff(vmode="integer", dim=c(10000,10000))
ffvecapply(Matrix[i1:i2] <- i1:i2, X=Matrix)

# split into tiles with 1000x1000 each,
# arranged in a metacube with 10x10 tiles
Tiles <- fftile(Matrix, ntile=c(10,10))
Tiles
Tiles[[2,3]]
# loop over Cublets
ProcTiles <- apply(Tiles, 1:2, function(x){
  list(rowSums(x[[1]][]))
})
ProcTiles[[1,1]]
ProcTiles <- fixapply(ProcTiles)
ProcTiles[[1,1]]
CollTiles <- fixapply(apply(ProcTiles, 1, function(x){
  list(do.call("csum", x))
}))
BindTiles <- unlist(CollTiles, use.names=FALSE)
BindTiles[1:10]
rowSums(Matrix[1:10,])
```



R.ff FUTURE …

finalizing as.ff()	in the making
implementing basic functions	in the making – help welcome
svd and friends?	Volunteers?
implementing stats & graphs	Volunteers?

… AND BEYOND

Source: Oehlschlägel, Adler, Nenadic, Zucchini (2008) A first glimpse into 'R.ff'

TEAM / CREDITS

package ff 1.0	**Daniel Adler, Oleg Nenadic, Walter Zucchini, Christian Gläser**
package ff 2.0	
Jens Oehlschlägel	**Jens_Oehlschlaegel@truecluster.com**
	R package design; Hybrid Index Preprocessing; transparent object creation and finalization; vmode design; virtualization and hybrid copying; arrays with dimorder and bydim; symmetric matrices; factors and POSIXct; virtual windows and transpose; new generics update, clone, swap, add, as.ff and as.ram; ffapply and collapsing functions. R-coding, C-coding and Rd-documentation.
Daniel Adler	**dadler@uni-goettingen.de**
	C++ generic file vectors, vmode implementation and low-level bit-packing/unpacking, arithmetic operations and NA handling, Memory-Mapping and backend caching modes. C++ coding and platform ports. R-code extensions for opening existing flat files readonly and shared.
package R.ff 0.1	
Jens Oehlschlägel	**Jens_Oehlschlaegel@truecluster.com**
	R package design; ff return value handling, ff function coercion, bigR, ffbatch, ffhash, bigorder, ffmatmul, ffmatinv, ffdist and virtual vdist, virtual cubelets