

Debugging

try
traceback, browser y
debug
recover
where
c, loops
t.race
warnings
debug de M. Bravington

Paralelización

Debugging código paralelizado

Debugging y
paralelización: una
práctica
Error handling y snowfall

ff y large objects

ff y paralelización

Curso de R avanzado. Debugging y paralelización

Ramón Díaz-Uriarte

Biología Computacional
Centro Nacional de Investigaciones Oncológicas (CNIO)
rdiaz02@gmail.com
<http://ligarto.org/rdiaz>
Copyright © 2010 Ramón Díaz-Uriarte

Curso de R Avanzado. Barcelona, 10-Diciembre-2010

Licencia y copyright



This work is Copyright, ©, 2011, Ramón Díaz-Uriarte, and is licensed under the **Creative Commons Attribution-NonCommercial-ShareAlike** License. To view a copy of this license, visit

<http://creativecommons.org/licenses/by-nc-sa/3.0/>

or send a letter to Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

Please, **respect the copyright**. This material is provided freely, and if you use it, I only ask that you use it according to the (very permissive) terms of the license: attribution, non-commercial use, and a share alike license. If you have any doubts, ask me.

Debugging

try
traceback, browser y
debug
recover
where
c, loops
trace
warnings
debug de M. Bravington

Paralelización

Debugging código paralelizado

Debugging y
paralelización: una
práctica
Error handling y snowfall

ff y large objects

ff y paralelización

Temario

Debugging

try
traceback, browser y
debug
recover
where
c, loops
t.r.a.c.e
warnings
debug de M. Bravington

Paralelización

Debugging código paralelizado

Debugging y
paralelización: una
práctica
Error handling y snowfall

ff y large objects

ff y paralelización

Debugging

try
traceback, browser y debug
recover
where
c, loops
t.r.a.c.e
warnings
debug de M. Bravington

Paralelización

Debugging código paralelizado

Debugging y paralelización: una práctica
Error handling y snowfall

ff y large objects

ff y paralelización

try

Debugging

try

traceback, browser y

debug

recover

where

c, loops

trace

warnings

debug de M. Bravington

Paralelización

Debugging

código

paralelizado

Debugging y
paralelización: una
práctica

Error handling y snowfall

ff y large objects

ff y paralelización

- *try* y *tryCatch* se comportan como otros exception handlers de otros lenguajes. *tryCatch* es algo más sofisticado pero, para muchos casos, *try* es suficiente.
- *withCallingHandlers* ofrece un mecanismo aun más general para capturar otras condiciones.
- *stopifnot* muy útil para “regression testing”.

try (cont.)

Debugging

try

traceback, browser y

debug

recover

where

c, loops

trace

warnings

debug de M. Bravington

Paralelización

Debugging

código

paralelizado

Debugging y

paralelización: una

práctica

Error handling y snowfall

ff y large objects

ff y paralelización

```
> ffail <- function(x, y) {
+   tf1 <- try(lm(y ~ x))
+   if (class(tf1) == "try-error")
+     print("Vaya, un error")
+   else print(summary(tf1))
+ }
> x <- rnorm(10)
> y <- rnorm(10)
> ffail(x, y)
> y <- 34
> ffail(x, y)
```

traceback

Debugging

try

traceback, browser y
debug

recover

where

c, loops

trace

warnings

debug de M. Bravington

Paralelización

Debugging

código

paralelizado

Debugging y
paralelización: una
práctica

Error handling y snowfall

ff y large objects

ff y paralelización

- Cuando se produce un **error**, `traceback()` nos informa de la secuencia de llamadas antes del crash de nuestra función. Util cuando se produce mensajes de error incomprensibles.

traceback (cont.)

Debugging

try

traceback, browser y
debug

recover

where

c, loops

trace

warnings

debug de M. Bravington

Paralelización

Debugging

código

paralelizado

Debugging y
paralelización: una
práctica

Error handling y snowfall

ff y large objects

ff y paralelización

```
> lm("a" ~ 99)
> traceback()
```

¿Dónde está el problema?

(Ojo: no si se hace con `try` o `tryCatch`)

browser y debug

Debugging

try

traceback, browser y
debug

recover

where

c, loops

trace

warnings

debug de M. Bravington

Paralelización

Debugging código

paralelizado

Debugging y
paralelización: una
práctica

Error handling y snowfall

ff y large objects

ff y paralelización

- *browser* interrumpe la ejecución a partir de ese punto y permite seguir la ejecución o examinar el entorno; con "n" paso a paso, si otra tecla sigue ejecución normal. "Q" para salir. "c" para ir al final del contexto (bucles y llamadas a funciones)
- *debug* es como poner un "browser" al principio de la función, y se ejecuta la función paso a paso. Se sale con "Q".

```
> debug(my.buggy.function)
```

```
> ...
```

```
> undebug(my.buggy.function)
```

Browser commands

(De Gentleman, “R programming for bioinformatics”, p. 275).

	Initial Mode	Step through Debugger Mode
n	start the step through debugger	execute the next step in the function
c	continue execution	continue execution; if inside a loop, execute until the loop ends
cont	same as c	same as c
carriage return	same as c	same as n

Table 9.2: Browser commands with modal functionalities.

browser y debug (cont.)

```
> my.f2 <- function(x, y) {  
+   z <- rnorm(10)  
+   y2 <- z * y  
+   y3 <- z * y * x  
+   return(y3 + 25)  
+ }
```

- Hagamos debugging

```
> my.f2(runif(3), 1:4)  
> debug(my.f2)  
> my.f2(runif(3), 1:4)  
> undebug(my.f2)
```

- Dentro del entorno de debugging podemos cambiar variables, crear variables, etc.
- Probémoslo.
- Podemos (re)definir funciones (y si existen fuera?)
- Y si tenemos un *debug* y hacemos un reload de ese código?

browser condicional

- Insertar un *browser* y correr de nuevo
- (Nótese también cambio en número que sigue a *Browse[...]*).
- En qué difieren *browser* y *debug*? Inicialmente, *browser* sólo interrumpe.
- Y si queremos un *browser* condicional? Ej., sólo si algún $y3 > 10$? Cómo lo haríais?

Debugging

try

traceback, browser y

debug

recover

where

c, loops

trace

warnings

debug de M. Bravington

Paralelización

Debugging

código

paralelizado

Debugging y

paralelización: una

práctica

Error handling y snowfall

ff y large objects

ff y paralelización

browser condicional

Debugging

try

traceback, browser y
debug

recover

where

c, loops

trace

warnings

debug de M. Bravington

Paralelización

Debugging

código

paralelizado

Debugging y
paralelización: una
práctica

Error handling y snowfall

ff y large objects

ff y paralelización

- Insertar un *browser* y correr de nuevo
- (Nótese también cambio en número que sigue a *Browse[...]*).
- En qué difieren *browser* y *debug*? Inicialmente, *browser* sólo interrumpe.
- Y si queremos un *browser* condicional? Ej., sólo si algún $y3 > 10$? Cómo lo haríais?

```
> my.f2 <- function(x, y) {  
+   z <- rnorm(10)  
+   y2 <- z * y  
+   y3 <- z * y * x  
+   if (any(y3 > 10))  
+     browser()  
+   return(y3 + 25)  
+ }  
> my.f2(8, 9)  
> my.f2(0, 8)
```

- (Otra opción, breakpoints condicionales, luego).

“on.exit(browser())”

Debugging

try

traceback, browser y
debug

recover

where

c, loops

trace

warnings

debug de M. Bravington

Paralelización

Debugging

código

paralelizado

Debugging y
paralelización: una
práctica

Error handling y snowfall

ff y large objects

ff y paralelización

```
> my.f2 <- function(x, y) {  
+   on.exit(browser())  
+   z <- rnorm(10)  
+   y2 <- z * y  
+   y3 <- z * y * x  
+   return(y3 + 25)  
+ }  
> my.f2(8, 9)  
> my.f2("a", 34)
```

debug en funciones de paquetes

Debugging

try

traceback, browser y

debug

recover

where

c, loops

trace

warnings

debug de M. Bravington

Paralelización

Debugging

código

paralelizado

Debugging y

paralelización: una

práctica

Error handling y snowfall

ff y large objects

ff y paralelización

- Cargamos un paquete de ejemplo y vemos qué hay dentro

```
> library(myRpack)
> ls(pos = 2)
> g1
> myRpack:::g2
> myRpack:::g3
```

- Podemos hacer debugging de código dentro

```
> debug(g3)
> debug(myRpack:::g3)
> g1()
> undebug(myRpack:::g3)
```

- `ls()`, etc.
- Si debugging `f1`, `debug(g3)` funciona (dentro de `g1`). Probad!
- Y ahora quitad el debugging code de `g3`.

Limitaciones *browser* y *debug*

Debugging

try

traceback, browser y
debug

recover

where

c, loops

trace

warnings

debug de M. Bravington

Paralelización

Debugging

código

paralelizado

Debugging y
paralelizacion: una
práctica

Error handling y snowfall

ff y large objects

ff y paralelización

- *browser*: peligroso dejarlo en nuestras funciones
- Cómo poneis un *browser* en *lm*?

Limitaciones *browser* y *debug*

Debugging

try

traceback, browser y
debug

recover

where

c, loops

trace

warnings

debug de M. Bravington

Paralelización

Debugging

código

paralelizado

Debugging y
paralelización: una
práctica

Error handling y snowfall

ff y large objects

ff y paralelización

- *browser*: peligroso dejarlo en nuestras funciones
- Cómo poneis un *browser* en *lm*?
- *browser*: no podemos introducirlo en código de paquetes.
- Y *debug*, aunque se puede usar con cualquier función, nos fuerza a ejecutar desde el principio. Necesitaremos algo más flexible.
- Y si sólo queremos entrar si se producen errores?

recover

Debugging

try
traceback, browser y
debug
recover
where
c, loops
trace
warnings
debug de M. Bravington

Paralelización

Debugging código paralelizado

Debugging y
paralelización: una
práctica
Error handling y snowfall

ff y large objects

ff y paralelización

- Podemos hacer que R, en uso interactivo, nos lleve a la función donde se genera el error de forma automática:

```
> opt <- options(error = recover)
> lm("a" ~ 99)
```

- Seleccionamos el “frame number”, y podemos ver los valores de variables locales, ejecutar código, etc.
- Para volver al comportamiento de siempre hacemos (por eso asignamos antes opt).

```
> options(opt)
```

- Pero sólo los objetos visibles desde la llamada particular (la que genera el error) son accesibles.
- ¿Qué objetos vemos desde los distintos frames?

recover (cont.)

Debugging

try

traceback, browser y

debug

recover

where

c, loops

trace

warnings

debug de M. Bravington

Paralelización

Debugging

código

paralelizado

Debugging y

paralelización: una

práctica

Error handling y snowfall

ff y large objects

ff y paralelización

Para el siguiente ejemplo:

- ¿Dónde se produce el error?
- Usar *traceback* primero
- Entremos en *f1*
- En *f2*: qué hay?
- ¿Dónde estamos?

recover (cont.)

```
> opt <- options(error = recover)
> f1 <- function(x) {
+   u1 <- 99
+   log(x)
+ }
> f2 <- function(z) {
+   u2 <- 89
+   ret <- 4 + f1(z)
+   cat("\n    algo que nos distrae\n")
+   t2 <- rnorm(2)
+   ret
+ }
> f3 <- function(y) {
+   u3 <- 5
+   cat("\n    el valor de u3 es ",
+       u3, "\n")
+   tmp <- f2(y) * 9
+   t3 <- 95
+   return(tmp)
+ }
> f3("a")
```

Debugging

try
traceback, browser y
debug
recover
where
c, loops
trace
warnings
debug de M. Bravington

Paralelización

Debugging código paralelizado

Debugging y
paralelización: una
práctica
Error handling y snowfall

ff y large objects

ff y paralelización

recover (cont.)

Debugging

try
traceback, browser y
debug
recover
where
c, loops
trace
warnings
debug de M. Bravington

Paralelización

Debugging código paralelizado

Debugging y
paralelización: una
práctica
Error handling y snowfall

ff y large objects

ff y paralelización

(Más de lo mismo, en un caso más sencillo)

```
> f4 <- function(x) {  
+   u4 <- 4  
+   if (x < 9)  
+     stop("ooo")  
+   x  
+ }  
> f5 <- function(v) {  
+   u5 <- 5  
+   f4(v)  
+ }  
> f5(12)  
> f5(3)  
> options(opt)
```

(Ojo: *recover* desde función en ejecución es muy confuso).

where

Debugging

try
traceback, browser y
debug
recover
where
c, loops
t.trace
warnings
debug de M. Bravington

Paralelización

Debugging código paralelizado

Debugging y
paralelización: una
práctica
Error handling y snowfall

ff y large objects

ff y paralelización

- *where* para ver el call stack
- (No mireis)
- ...
- ¿Dónde estoy? (Entrad hasta $f1$, via “n”, y luego id saliendo de nuevo).
- *where* sirve para “orientarse”

```
> debug (f3)
> debug (f2)
> debug (f1)
> f3 (19)
> undebug (f1)
> undebug (f2)
> undebug (f3)
```

C, loops, etc

Debugging

try
traceback, browser y
debug
recover
where
c, loops
trace
warnings
debug de M. Bravington

Paralelización

Debugging código paralelizado

Debugging y
paralelización: una
práctica
Error handling y snowfall

ff y large objects

ff y paralelización

```
> f7 <- function(x) {  
+   fi <- function(z, s) {  
+     uu <- z + 3  
+     vv <- uu + 9  
+     vv * s  
+   }  
+   if (x > 5) {  
+     cat("\n inside x > 5\n")  
+     mm <- x * 3  
+     mm2 <- sqrt(mm)  
+   }  
+   for (i in 1:5) {  
+     print(fi(x, i))  
+   }  
+   print("\n al final del loop")  
+   x  
+ }  
> debug(f7)  
> f7(8)
```

c, loops, etc (cont.)

Debugging

try
traceback, browser y
debug
recover
where
c, loops
trace
warnings
debug de M. Bravington

Paralelización

Debugging código paralelizado

Debugging y
paralelización: una
práctica
Error handling y snowfall

ff y large objects

ff y paralelización

- Desde mitad de código, llegad al final (ej., justo cuando hace *inside x*).. Usad *c* en *f7*, no dentro loop
- Usad *c* para saltar el loop (sólo una vez iniciado el *for*)
- Entrad, poned *debug(fi)*, usad *c* para saltar evaluaciones de *fi*
- (Pero si debugging *fi*, el *c* salta evaluación de *fi*, no del loop)

trace

Debugging

try
traceback, browser y
debug
recover
where
c, loops
t.trace
warnings
debug de M. Bravington

Paralelización

Debugging código paralelizado

Debugging y
paralelización: una
práctica
Error handling y snowfall

ff y large objects

ff y paralelización

- `trace` tiene múltiples usos, algunos complicados
- Insertar llamadas a `browser`. Una de las mejores formas en código de paquetes, etc
- Si queremos ir paso a paso, teclear “n” en cuanto aparezca `Browse`.

```
> trace(lm, tracer = browser)
> lm(5 ~ 6)
> untrace(lm)
```

- Y podemos “pillar” funciones que son llamadas por otras.
- (Probad y usad `where`, modificad valores, etc).

```
> trace(f1, tracer = browser)
> f3(13)
> untrace(f1)
```

trace (cont.)

Debugging

try
traceback, browser y
debug
recover
where
c, loops
t r a c e
warnings
debug de M. Bravington

Paralelización

Debugging código paralelizado

Debugging y
paralelización: una
práctica
Error handling y snowfall

ff y large objects

ff y paralelización

- Podemos incluir otros “tracers”

```
> trace(f1, tracer = recover)
> f3(16)
> untrace(f1)
```

- Entrad en los tres sitios, ved qué hay, etc.
- Entrad en los tres sitios, y teclad n (y luego 0). ¿Qué ocurre?
- (Rara vez lo uso, pero bueno saber que existe).

trace en líneas concretas

Debugging

try
traceback, browser y
debug
recover
where
c, loops
trace
warnings
debug de M. Bravington

Paralelización

Debugging código paralelizado

Debugging y
paralelización: una
práctica
Error handling y snowfall

ff y large objects

ff y paralelización

- Podemos incluir la línea
- Y cómo identificamos la línea que nos interesa?

```
> as.list(body(f3))  
> trace(f3, tracer = browser,  
+       at = 5)  
> f3  
> body(f3)  
> f3(16)  
> untrace(f3)
```

- `ls()`, teclad `n`, etc.

trace condicionales

Debugging

try
traceback, browser y
debug
recover
where
c, loops
trace
warnings
debug de M. Bravington

Paralelización

Debugging código paralelizado

Debugging y
paralelización: una
práctica
Error handling y snowfall
ff y large objects
ff y paralelización

- ... y podemos hacerlo condicional

```
> as.list(body(f3))  
> trace(f3, tracer = quote(if (y >  
+     20) {  
+     browser()  
+ })), at = 5)  
> f3  
> body(f3)  
> f3(16)  
> f3(24)  
> untrace(f3)
```

- En vez de *browser* podríamos poner *recover*. U otras cosas.

trace y edit

Debugging

try
traceback, browser y
debug
recover
where
c, loops
trace
warnings
debug de M. Bravington

Paralelización

Debugging código paralelizado

Debugging y
paralelización: una
práctica
Error handling y snowfall

ff y large objects

ff y paralelización

- Y si queremos insertar código arbitrario?
- Podemos modificar funciones de paquetes!!
- Incluidas no exportadas

```
> trace(lm, edit = TRUE)
> untrace(lm)
> trace(g3, edit = TRUE)
> trace(myRpack:::g3, edit = TRUE)
```

- Insertad algo en `g3`

```
> body(myRpack:::g3)
> g1()
> untrace(myRpack:::g3)
> body(myRpack:::g3)
```

trace y edit (cont.)

Debugging

try
traceback, browser y
debug
recover
where
c, loops
trace
warnings
debug de M. Bravington

Paralelización

Debugging código paralelizado

Debugging y
paralelización: una
práctica
Error handling y snowfall

ff y large objects

ff y paralelización

- Podemos repetir cuantas veces queramos
- Si estamos editando una función de un paquete, los cambios se pierden al salir de R y, por eso, es fundamental guardar el código de la función que estamos editando. Si es una función definida en el “global environment”, los cambios se preservan si salvamos el global environment (por ej., con un `save.image`).

warnings, debugging

Debugging

try
traceback, browser y
debug
recover
where
c, loops
trace
warnings
debug de M. Bravington

Paralelización

Debugging código paralelizado

Debugging y
paralelización: una
práctica
Error handling y snowfall

ff y large objects

ff y paralelización

Una opción: warnings a errores

```
> opt <- options(error = recover)
> options(warn = 2)
> f3(-9)
> options(opt)
> options(warn = 0)
```

- (Cambio opciones: “si de verdad”: `opt <- options(error = recover, warn = 2)`).
- Ojo con ir dejándonos browsers, tracing, opts, etc. Puede ser confuso.

warnings, debugging

Debugging

try

traceback, browser y

debug

recover

where

c, loops

trace

warnings

debug de M. Bravington

Paralelización

Debugging

código

paralelizado

Debugging y

paralelización: una
práctica

Error handling y snowfall

ff y large objects

ff y paralelización

Otra opción: usar `trace` con `recover` o `browser` (limitado: no funciona si warning desde código en C). No lo encuentro muy útil.

```
> trace(warning, recover)
> f33 <- function(x) {
+   if( x < 3)
+     warning("un warning")
+ }
> f33(2)
> f3(-9)
> untrace(warning)
> ## Y esto qué hace? Teclead "n"
> trace(warning, browser)
> f33(2)
> untrace(warning)
```

Paquete *debug*

Debugging

try
traceback, browser y
debug
recover
where
c, loops
t r a c e
warnings
debug de M. Bravington

Paralelización

Debugging código paralelizado

Debugging y
paralelización: una
práctica
Error handling y snowfall

ff y large objects

ff y paralelización

- Alternativa a *trace* y *browser*
- Poder seguir corriendo a pesar errores
- Posibilidad mover punto ejecución (saltar a sitios)
- No funciona del todo bien con ESS (una chapucilla: cambiar a *shell-mode*, y luego volver a *inferior-ess-mode*).

Paquete *debug* (cont.)

Debugging

try
traceback, browser y
debug
recover
where
c, loops
trace
warnings
debug de M. Bravington

Paralelización

Debugging código paralelizado

Debugging y
paralelización: una
práctica
Error handling y snowfall

ff y large objects

ff y paralelización

- *mtrace*: que se haga debugging
- *qqq()*: salir
- *mtrace(function, tracing = FALSE);*
mtrace.off(): quitar debugging a todas
- *bp(line)*: breakpoint en esa line
- *skip(line)*: saltar a esa línea; *skip(1)* re-iniciar
- *go()*: continuar hasta siguiente breakpoint
- Confuso? funciones en namespaces, pero no visibles

Paquete *debug* (cont.)

Debugging

try
traceback, browser y
debug
recover
where
c, loops
t race
warnings
debug de M. Bravington

Paralelización

Debugging código paralelizado

Debugging y
paralelización: una
práctica
Error handling y snowfall

ff y large objects

ff y paralelización

Problemas con errores, y los comandos de arriba: *go*,
skip, *bp*

```
> library(debug)
> mtrace(g3)
> g3
> myRpack:::g3
> g1()
> mtrace.off()
> mtrace(g1)
> g1
> g1()
> mtrace(g2)
> g1()
> mtrace(g1)
> g1()
> g1(-1)
```

Paquete *debug* (cont.)

Debugging

try
traceback, browser y
debug
recover
where
c, loops
trace
warnings
debug de M. Bravington

Paralelización

Debugging código paralelizado

Debugging y
paralelización: una
práctica
Error handling y snowfall

ff y large objects

ff y paralelización

(Un ejercicio)

- Usad función `f7`
- breakpoint en línea 6
- id allí directamente: ¿qué pasa?

Paquete *debug* (cont.)

Debugging

try
traceback, browser y
debug
recover
where
c, loops
trace
warnings
debug de M. Bravington

Paralelización

Debugging código paralelizado

Debugging y
paralelización: una
práctica
Error handling y snowfall

ff y large objects

ff y paralelización

- breakpoints condicionales una vez iniciado el debugging
- incluso antes de empezar `mtrace`, si tercer argumento

```
> mtrace(g1)
> bp(3, min(x) < 0, "g1")
> g1() ## dentro: haced go()
> g1(-5) ## dentro: haced go()
```

- (No se ve el breakpoint conditional code.)
- (Cambiad x antes de llegar al breakpoint: realmente es condicional)

Paralelización: intro

Debugging

try
traceback, browser y
debug
recover
where
c, loops
t. race
warnings
debug de M. Bravington

Paralelización

Debugging código paralelizado

Debugging y
paralelización: una
práctica
Error handling y snowfall

ff y large objects

ff y paralelización

- ¿Por qué?
- Embarazosamente paralelizable
- Lógica: un master, varios slaves
- MPI, PVM, sockets, NWS
- Rmpi, rpvm, nws
- snow, snowfall

Notas

Debugging

try
traceback, browser y
debug
recover
where
c, loops
trace
warnings
debug de M. Bravington

Paralelización

Debugging código paralelizado

Debugging y
paralelización: una
práctica
Error handling y snowfall

ff y large objects

ff y paralelización

- PVM: poco uso. Entre MPI y PVM la elección para primer uso creo está clara
- MPI: ¿qué MPI?
- Rmpi y papply: bonita combinación. Bajo nivel si quiero, alto cuando quiero. Fácil debugging (por papply). papply: no longer maintained
- snow: demasiado farragoso para mi. Muchas “capas intermedias”, y error-handling no me gusta
- MPI: configuración. Sockets funcionan en (casi) todas partes
- snowfall: lo hace todo (a veces demasiado) sencillo
- NWS: tentadora la intercomunicación de lenguajes y el modelo. ¿Overhead del setup?
- Ganancia: en buenos casos, casi lineal con número de CPUs.
- Otros paquetes para R, como *foreach* y similares.

Consideraciones

Debugging

try
traceback, browser y
debug
recover
where
c, loops
t.race
warnings
debug de M. Bravington

Paralelización

Debugging código paralelizado

Debugging y
paralelización: una
práctica
Error handling y snowfall

ff y large objects

ff y paralelización

- Número de nodos esclavos
- Almacenamiento común y acceso
- Relación entre cálculo, envío de datos, I/O
- Tolerancia a fallos
- Load-balancing
- Sólo ejemplos con *snowfall* aquí

Ejemplos con snowfall

Debugging

try
traceback, browser y
debug
recover
where
c, loops
trace
warnings
debug de M. Bravington

Paralelización

Debugging código paralelizado

Debugging y
paralelización: una
práctica
Error handling y snowfall

ff y large objects

ff y paralelización

Primero ejemplos, y números aleatorios

```
> sfInit(parallel = TRUE, cpus = 4, type = "SOCK")
> ## No usar lo que dicen en tutorial, p. 10
> sfClusterSetupRNG()
> sfClusterApply(1:4, rnorm)
> sfClusterApply(1:4, function(x) rnorm(1))
> sfClusterApply(1:4, function(x) rnorm(x, sd = 10))
> sfClusterApply(1:4, rnorm, sd = 10)
> sfClusterApply(1:4, function(x) rnorm(1))
> sfClusterApply(1:4, function(x) rnorm(1, sd = 10))
> sfClusterApply(1:4, rnorm(1, sd = 10))
> sfClusterSetupRNG(seed = rep(12, 6))
> sfSapply(1:8, function(x) rnorm(1))
> sfClusterApply(1:8, function(x) rnorm(1))
```

Ejemplos con snowfall (cont.)

Debugging

try
traceback, browser y
debug
recover
where
c, loops
t race
warnings
debug de M. Bravington

Paralelización

Debugging código paralelizado

Debugging y
paralelización: una
práctica
Error handling y snowfall

ff y large objects

ff y paralelización

```
> sfClusterSetupRNG(seed = rep(12, 6))
> sfClusterApplyLB(1:8, function(x) rnorm(1))
> sfClusterSetupRNG(seed = rep(12, 6))
> unlist(sfClusterApplyLB(1:8, function(x) rnorm(1)))
> sfInit(parallel = TRUE, cpus = 8, type = "SOCK")
> sfClusterSetupRNG(seed = rep(12, 6))
> sfSapply(1:8, function(x) rnorm(1))
> sfClusterSetupRNG(seed = rep(12, 6))
> unlist(sfClusterApplyLB(1:8, function(x) rnorm(1)))
```

Ejemplos con snowfall (cont.)

Exportar objetos o pasar en argumentos

```
> sfClusterEval(rnorm(3))
> sfClusterCall(rnorm, 3)
> sfClusterCall(function() rnorm(3))
> sfClusterEval(ls())
> sfClusterCall(function() ls())
> x1 <- 1:10
> x2 <- 11:20
> sfExport("x1", "x2")
> ## sfExportAll???
>
> sfClusterEval(ls())
> sfClusterEval(x1)
> sfClusterApplyLB(100:110, sum, x1)
> sfClusterApplyLB(1:10, function(i) {x1[i] * x2[i]})
> ## Y si cambio x2?
> x2 <- 1000:1010
> sfClusterApplyLB(1:10, function(i) {x1[i] * x2[i]})
> ## x3 pasado como argumento
> x3master <- 11:20
> sfClusterApplyLB(1:10,
+                   function(i, x3) {x1[i] * x3[i]},
+                   x3 = x3master)
```

Ejemplos con snowfall (cont.)

Debugging

try
traceback, browser y
debug
recover
where
c, loops
trace
warnings
debug de M. Bravington

Paralelización

Debugging código paralelizado

Debugging y
paralelización: una
práctica
Error handling y snowfall

ff y large objects

ff y paralelización

(Del tutorial de Knaus y Porzelius)

```
> sfInit(parallel = TRUE, cpus = 4, type = "SOCK")
> sfLibrary(randomForest)
> data(iris)
> boot.n <- 5 ## for real, much larger!!
> boot.index <-
+   t(sapply(1:boot.n,
+           function(b){ sample(1:nrow(iris),
+                               replace=TRUE) })))
> not.in.sample <- list()
> for (i in 1:boot.n){
+   not.in.sample[[i]] <-
+     (1:nrow(iris))[-unique(boot.index[i, ])]
+ }
```

Ejemplos con snowfall (cont.)

(Del tutorial de Knaus y Porzelius, cont.)

```
> boot.fun <- function(actual.sample){
+   sample.fit <-
+     randomForest(Species ~ .,
+                 data =
+                   iris[boot.index[actual.sample, ], ])
+   prediction <-
+     predict(sample.fit,
+            newdata =
+              iris[not.in.sample[[actual.sample]],
+                  1:4])
+   sum(prediction !=
+        iris[not.in.sample[[actual.sample]], 5])/
+     length(prediction)
+ }
> sfExport("iris", "boot.index", "not.in.sample")
> sfClusterEval(ls())
> (res <- sfClusterApplyLB(1:boot.n, boot.fun))
> (error <- mean(unlist(res)))
> ## Resultados son listas.
> ## Casi siempre procesado adicional necesario
```

Debugging

try
traceback, browser y
debug
recover
where
c, loops
trace
warnings
debug de M. Bravington

Paralelización

Debugging
código
paralelizado

Debugging y
paralelización: una
práctica
Error handling y snowfall

ff y large objects
ff y paralelización

Debugging código paralelizado

Debugging

try
traceback, browser y
debug
recover
where
c, loops
t.race
warnings
debug de M. Bravington

Paralelización

Debugging código paralelizado

Debugging y
paralelización: una
práctica
Error handling y snowfall

ff y large objects

ff y paralelización

- Mismas ideas generales
- No acceso directo a nodos esclavos
 - ▶ Que devuelvan errores al master. Rara vez inteligible con snow/snowfall. Crear nuestra clase try.
 - ▶ Replicar en master lo que pasa en el esclavo

Un error habitual: ¿dónde está la función?

Debugging

try
traceback, browser y
debug
recover
where
c, loops
trace
warnings
debug de M. Bravington

Paralelización

Debugging código paralelizado

Debugging y
paralelización: una
práctica
Error handling y snowfall

ff y large objects

ff y paralelización

```
> fs1 <- function(x) {  
+   3 * x  
+ }  
> sfClusterApplyLB(1:5, fs1)  
> sfClusterApplyLB(1:5,  
+   function(x) fs1(9))  
> sfExport("fs1")  
> sfClusterApplyLB(1:5, function(x) fs1(9))
```

Debugging código paralelizado (cont.)

Debugging

try
traceback, browser y
debug
recover
where
c, loops
trace
warnings
debug de M. Bravington

Paralelización

Debugging código paralelizado

Debugging y
paralelización: una
práctica
Error handling y snowfall

ff y large objects

ff y paralelización

```
> h1 <- function(x, y) {  
+   tmp <- log(y)  
+   x <- 2 * x  
+   rnorm(x, sd = tmp)  
+ }  
> sfInit(parallel = TRUE, cpus = 4, type = "SOCK")  
> sfClusterEval(ls()) ## no hay nada  
> ## Este mensaje ayuda poco  
> xn <- c(1, 2, 3, 2, -3, 2)  
> sfClusterApplyLB(xn,  
+                   h1,  
+                   y = 3)  
> ## Y si en serie con snowfall?  
> sfInit(parallel = FALSE)  
> ## Algo mejor, pero no mucho  
> sfClusterApplyLB(xn,  
+                   h1,  
+                   y = 3)
```

Debugging código paralelizado (cont.)

Debugging

try
traceback, browser y
debug
recover
where
c, loops
trace
warnings
debug de M. Bravington

```
> ## Forzar ejecución uno a uno en master. Sencillo!!  
> for(i in xn) {  
+   cat("\n Haciendo xn ", i, "\n")  
+   h1(i, 3)  
+ }
```

Paralelización

Debugging código paralelizado

Debugging y
paralelización: una
práctica
Error handling y snowfall

ff y large objects

ff y paralelización

```
> ## Aha! es el xn = -3. Ver qué pasa ahí  
> h1(-3, 3)  
> ##  
> ## Si sospecho algunos, restringir "i" a esos  
> for(i in xn[4:6]) {  
+   cat("\n Haciendo xn ", i, "\n")  
+   h1(i, 3)  
+ }
```

(Todo lo anterior puede también hacerse dentro de un entorno de debugging).

Debugging y paralelización: una práctica

Debugging

try
traceback, browser y
debug
recover
where
c, loops
trace
warnings
debug de M. Bravington

Paralelización

Debugging código paralelizado

Debugging y
paralelización: una
práctica
Error handling y snowfall
ff y large objects
ff y paralelización

Usad tanto funciones habituales como *mtrace* (en paquete debug)

```
> sfInit(parallel = TRUE, cpus = 4, type = "SOCK")  
> library(myRpack)  
> g1(-1, snowfall = TRUE)
```

(*g2p* llama a *g3* en maestro. Y si hiciera

sfClusterApplyLB(x, function(z) g3(z))?

Recordad *sfLibrary*? Por qué sí o por qué no aquí?).

(MUCHO OJO con lo que pasa dentro de *g1*, *g3*, etc!!).

Una pista: podría estar enviándose a los nodos todo el environment?).

Según la documentación ...

Debugging

try
traceback, browser y
debug
recover
where
c, loops
trace
warnings
debug de M. Bravington

Paralelización

Debugging código paralelizado

Debugging y
paralelización: una
práctica

Error handling y snowfall

ff y large objects

ff y paralelización

- P. 6 en viñeta snowfall: *stopOnError* (que no se usa en *clusterApplyLB*, *sfSapply*, *sfLapply*)
- Usando *options(error = ...)*
- Lo primero no parece funcionar donde queremos
- Lo segundo?

```
> opt <- options(error = recover)
> sfClusterApplyLB(xn,
+   h1, y = 3)
> options(opt)
```

- En lo que sigue: ilustro una forma quizás heterodoxa

¿Qué hace snowfall si hay errores?

Debugging

try
traceback, browser y
debug
recover
where
c, loops
trace
warnings
debug de M. Bravington

Paralelización

Debugging código paralelizado

Debugging y
paralelización: una
práctica

Error handling y snowfall

ff y large objects

ff y paralelización

- Qué hace snowfall con los errores?

```
> clusterApply  
> clusterApplyLB  
> staticClusterApply  
> dynamicClusterApply  
> checkForRemoteErrors
```

- snowfall: Wrapps las llamadas en `try` y mira la clase
- Intentemos evitar que snowfall detecte el error antes que nosotros. Nosotros hacemos el error handling.

Modificando clase `try-error`

Debugging

try
traceback, browser y
debug
recover
where
c, loops
trace
warnings
debug de M. Bravington

Paralelización

Debugging código paralelizado

Debugging y
paralelización: una
práctica

Error handling y snowfall

ff y large objects

ff y paralelización

```
> h2 <- function(x, y) {  
+   tmpf <- try(h1(x, y))  
+   if(inherits(tmpf, "try-error")) {  
+     class(tmpf) <- "my-try-error"  
+     ## otras cosas?  
+   }  
+   return(tmpf)  
+ }  
> ## Podríamos tb. definir h1 inside h2 y evitar:  
> sfExport("h1")  
> ## Be explicit where things live  
> rm(h1)  
> sfClusterApplyLB(xn, h2,  
+                   y = 3)  
> outsf <- sfClusterApplyLB(xn, h2,  
+                             y = 3)  
> any(unlist(lapply(outsf,  
+                   function(x)  
+                     inherits(x, "my-try-error"))))
```

¿Es necesario?

Debugging

try
traceback, browser y
debug
recover
where
c, loops
t r a c e
warnings
debug de M. Bravington

Paralelización

Debugging código paralelizado

Debugging y
paralelización: una
práctica

Error handling y snowfall

ff y large objects

ff y paralelización

- Para un uso de pocas veces, o por nosotros mismos, probablemente no.
- Si un paquete: necesitamos algo robusto, y lo anterior puede servir.
- Permite afrontar tipos de errores que podemos anticipar.

ff y large objects

Debugging

try
traceback, browser y
debug
recover
where
c, loops
t r a c e
warnings
debug de M. Bravington

Paralelización

Debugging código paralelizado

Debugging y
paralelización: una
práctica
Error handling y snowfall

ff y large objects

ff y paralelización

- Tema muy largo
- Intro en p. 136 documento de Dirk Eddelbuettel
- Más información en ayuda de paquete *ff* y “ff-examples-concepts.pdf”
- *ff*: en R (y en RAM) sólo un puntero de objetos arbitrariamente grandes que se guardan en disco (*bigmemory* usa RAM, aunque fuera de R).

Paquete *ff*

Debugging

try
traceback, browser y
debug
recover
where
c, loops
trace
warnings
debug de M. Bravington

Paralelización

Debugging código paralelizado

Debugging y
paralelización: una
práctica
Error handling y snowfall

ff y large objects

ff y paralelización

- Creación de objetos similares a vectores, matrices, data frames
- Funciones tipo “apply” optimizadas para no agotar memoria (*?ffapply*)
- Muchas opciones; paquete complejo
- Ojo con qué y dónde se guarda (*ffsave*, opción *pattern* en *ff*, *as.ffdf*, etc)
- Procesar por “cachos”, cada uno de los cuales cabe en R

Ejemplo en docs. *ff*

Debugging

try
traceback, browser y
debug
recover
where
c, loops
trace
warnings
debug de M. Bravington

Paralelización

Debugging código paralelizado

Debugging y
paralelización: una
práctica
Error handling y snowfall

ff y large objects

ff y paralelización

- Muchas posibles opciones
- Ver experimentos en paquete experimental *R.ff* (http://r-forge.r-project.org/R/?group_id=483). Instalar como

```
install.packages("R.ff",  
repos="http://R-Forge.R-project.org").
```
- Este ejemplo (modificado) de “ff-examples-concepts.pdf”

Ejemplo en docs. (cont.)

Debugging

try
traceback, browser y
debug
recover
where
c, loops
trace
warnings
debug de M. Bravington

Paralelización

Debugging código paralelizado

Debugging y
paralelización: una
práctica
Error handling y snowfall

ff y large objects

ff y paralelización

```
> tmp <- vector("numeric", length = 1e7)
> object.size(tmp)
> rm(tmp)
> gc()
> library(ff)
> fd <- ff(vmode="double", length=1e7)
> object.size(fd)
> filename(fd)
> fd
> library(snowfall)
> sfInit(parallel = TRUE, cpus = 4, type = "SOCK")
> finalizer(fd)
> # let slaves not delete fd on shutdown
> finalizer(fd) <- "close" ## necesario?
> sfLibrary(ff)
> # do not export the same ff multiple times
> sfExport("fd")
> # explicitly opening avoids a gc problem
> sfClusterEval(open(fd))
```

Ejemplo en docs. (cont.)

Debugging

try
traceback, browser y
debug
recover
where
c, loops
trace
warnings
debug de M. Bravington

Paralelización

Debugging código paralelizado

Debugging y
paralelización: una
práctica
Error handling y snowfall

ff y large objects ff y paralelización

```
> chunk(fd)
> ## ojo lo que hace: modifica el objeto!
> system.time(
+   sfLapply( chunk(fd), function(i){
+     fd[i] <- runif(sum(i))
+     invisible()
+   })
+ )
> ## qué hace?
> sum(chunk(fd)[[1]])
> sum(chunk(fd)[[3]])
> system.time(
+   s <- sfLapply( chunk(fd),
+     function(i) quantile(fd[i], c(0.05, 0.95)) )
+ )
> # for completeness
> sfClusterEval(close(fd))
> csummary(s)
```

ff y paralelización: ADaCGH2

Debugging

try
traceback, browser y
debug
recover
where
c, loops
trace
warnings
debug de M. Bravington

Paralelización

Debugging código paralelizado

Debugging y
paralelización: una
práctica
Error handling y snowfall

ff y large objects

ff y paralelización

- Una solución más compleja (para problema más complejo)
- Argumento:
 - ▶ si `ff` guarda el objeto en disco
 - ▶ si puedo acceder a (distintas partes de) ese objeto (lectura sólo!) desde varios procesos
 - ▶ puedo procesar distintas partes de ese objeto a la vez
 - ▶ cada nodo puede escribir cosas distintas en objetos `ff` distintos
- Precauciones y similares
 - ▶ Lentitud de I/O (ej., NFS) y looking
 - ▶ Lectura sólo!
- Usado con éxito en nuevo paquete *ADaCGH*, la base de nuestra aplicación WaviCGH (<http://wavi.bioinfo.cnio.es>).

ff y paralelización: ADaCGH2 (cont.)

Debugging

try
traceback, browser y
debug
recover
where
c, loops
trace
warnings
debug de M. Bravington

Paralelización

Debugging código paralelizado

Debugging y
paralelización: una
práctica
Error handling y snowfall

ff y large objects

ff y paralelización

- Leer datos y crear objetos *ff*:
inputDataToADaCGHData
- (Uso de *multicore*)
- Funciones para acceder a objeto *ff* en los nodos:
getffObj. Sólo hace el objeto disponible
- Funciones para acceder a una parte de los datos:
getCGHvalue, *etc*. Accede a una parte
determinada del *ff.df*, que se pasa como índice.
Cierra objeto *ff* antes de devolver datos.
- Llamada a snowfall: *clusterApplyLB(indice, funcion, otros)*. *indice*: la parte de datos (*ff.df*) que lee *funcion*.

ff y paralelización: ADaCGH2 (cont.)

Debugging

try
traceback, browser y
debug
recover
where
c, loops
trace
warnings
debug de M. Bravington

Paralelización

Debugging código paralelizado

Debugging y
paralelización: una
práctica
Error handling y snowfall

ff y large objects

ff y paralelización

- En cada run en cada proceso se llama a *getffObj*, y después a *getCGHvalue*
- Cada run en un proceso esclavo SOLO carga el cacho de “índice”
- Cada nodo escribe un objeto *ff* (ej., *ffListOut*). Son objetos distintos en cada run*nodo.
- En nodo maestro, se compone un objeto *ffdf* a partir de los *ff* individuales (*outToffdf*).
- Ejemplos: *pSegmentHaarSeg*, *pSegmentGLAD*