HPC Aplicado con R

George G. Vega Yon

Contents

[1] Código copiado literalmente del paquete R `profvis` aquí.

Prefacio

El lenguaje de programación R [1] puede ser fantástico para la mayoría de las tareas diarias. Pero tan pronto como comienzas a lidiar con problemas más complicados, puedes enfrentarte al cuello de botella del bucle for. Si alguna vez te encuentras con tal problema, este libro es para ti. HPC Aplicado con R es una colección de charlas y conferencias que he dado sobre cómo acelerar tu código R usando computación paralela y otros recursos como C++. Los contenidos se han desarrollado principalmente durante mi tiempo en USC y UofU.[2]

El libro fue escrito usando quarto y está alojado en GitHub, donde puedes acceder a todo el código fuente.

Sobre el Autor

Soy Profesor Asistente de Investigación en la **División de Epidemiología de la Universidad de Utah**, donde trabajo estudiando Sistemas Complejos utilizando Computación Estadística. Nací y crecí en Chile. Tengo más de diez años de experiencia desarrollando software científico con enfoque en computación de alto rendimiento, visualización de datos y análisis de redes sociales. Mi formación es en Políticas Públicas (M.A. UAI, 2011), Economía (M.Sc. Caltech, 2015), y Bioestadística (Ph.D. USC, 2020).

Obtuve mi Ph.D. en Bioestadística bajo la supervisión del **Prof. Paul Marjoram** y la **Prof. Kayla de la Haye**, con mi disertación titulada "*Essays on Bioinformatics and Social Network Analysis: Statistical and Computational Methods for Complex Systems.*"

Si deseas aprender más sobre mí, por favor visita mi sitio web en https://ggvy.cl.

Sobre la versión en Español

Esta versión en español del libro fue creada para hacer accesible el contenido sobre computación de alto rendimiento con R a la comunidad hispanohablante. Aunque se ha hecho un esfuerzo por mantener la precisión técnica y el contexto, algunos términos especializados pueden requerir revisión adicional. La versión original en inglés permanece como la referencia autorizada.

Los lectores que encuentren errores de traducción o áreas que requieran clarificación son bienvenidos a contribuir reportando problemas en el repositorio de GitHub del proyecto.

Divulgación sobre IA

A partir de mediados de 2023, he estado utilizando IA para ayudarme a escribir este libro. Principalmente, uso una combinación de GitHub co-pilot, que ayuda con código y texto. El papel de la IA ha sido ayudarme a escribir más rápido y con mayor precisión, pero no ha estado involucrada en la conceptualización del libro o el desarrollo de los métodos presentados aquí.

Sobre la portada

La imagen de portada fue creada usando ChatGPT/Dall-E versión 5 usando el prompt "Necesito que dibujes una imagen que represente computación de alto rendimiento usando el lenguaje de programación R (que se usa principalmente para estadística). La imagen irá como portada del libro"Applied HPC with R". Debería incluir componentes relacionados con C++, computación paralela, y computación de alto rendimiento"

[2] Con muchos a quienes agradecer, incluyendo Paul Marjoram, Zhi Yang, Emil Hvitfeldt, Malcolm Barrett, Garrett Weaver, el grupo de investigación IMAGE P01 de USC, y mis estudiantes tanto en USC como en UoU.

I Fundamentos

[3]Código copiado literalmente del paquete R `profvis` aquí.

1. Perfilado de Código

> ⚠ Nota de Traducción
>
> Esta versión del capítulo fue traducida de manera automática utilizando IA. El capítulo aún no ha sido revisado por un humano.

El perfilado de código es una herramienta fundamental para programadores. Con el perfilado, es posible **identificar posibles cuellos de botella en tu código**, así como uso excesivo de memoria. Algunas cosas importantes a considerar al perfilar tu código:

- **Tiene un componente estocástico**: Independientemente de la complejidad computacional, un programa puede exhibir diferentes características de rendimiento en diferentes ejecuciones. Esto significa que siempre debes perfilar tu código múltiples veces y tomar el rendimiento promedio como resultado final.
- **Es inútil si el programa es demasiado rápido**: La mayoría del tiempo, el perfilado de código debe hacerse en programas que toman una cantidad notable de tiempo para ejecutarse. Si un programa se ejecuta demasiado rápido, la sobrecarga del perfilado mismo puede sesgar los resultados. Si aún necesitas abordar esto, puedes, por ejemplo, hacer múltiples ejecuciones o usar un conjunto de datos más grande para aumentar el tiempo de ejecución del programa.
- **No lo hagas demasiado temprano**: La mayoría del tiempo, los desarrolladores tienden a optimizar (y por tanto perfilar) su código demasiado temprano. No quieres gastar tiempo haciendo perfilado de código en algo que podría cambiar rápidamente. Por lo tanto, es mejor hacerlo en código que funciona en lugar de en piezas del mismo.

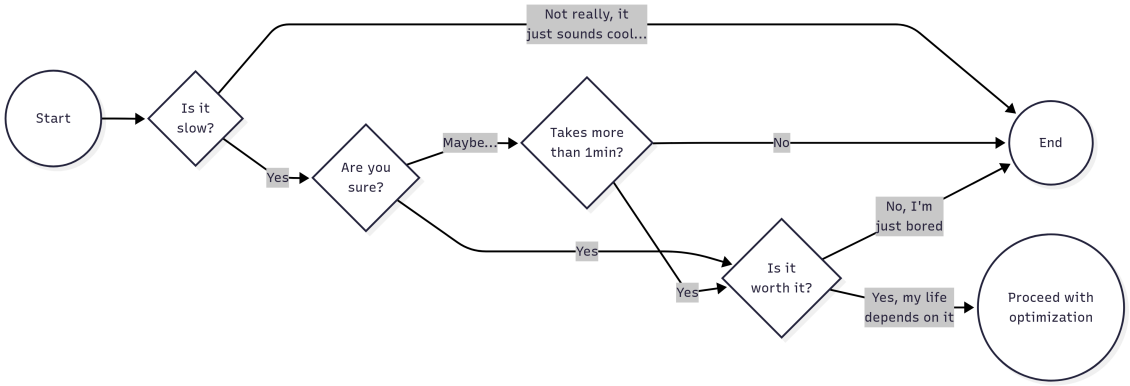
  En lugar de perfilar el código demasiado temprano, asegúrate de tener un buen plan de diseño e implementación.
- **Sé estratégico**: No todo necesita ser optimizado. Enfócate en las partes del código que son críticas para el rendimiento y la experiencia del usuario. Puedes pensar en términos de cuánto tiempo el programa/usuario gasta en cada parte.
- **La IA puede darte buenos consejos**: En mi experiencia personal, la IA puede ser útil para hacer perfilado de código también. Esto funciona mejor si estás usando una IA agéntica, ya que es más probable obtener buena retroalimentación de ella (probará el código por ti).

Aquí hay un flujo de trabajo propuesto para perfilar y optimizar tu código en general

1.1 Un flujo de trabajo propuesto

Aquí hay una fórmula a seguir cuando hagas perfilado/optimización de código:

1. Hazte estas preguntas antes de saltar al perfilado:



2. Si tienes éxito, entonces asegúrate de que el perfilado se haga en un tiempo finito, esto es, usa un subconjunto de los datos para evitar esperas largas. **Ejecutar el perfilador agregará tiempo de cómputo adicional, así que trata de mantenerlo corto (p. ej., 1 minuto)**.
3. Puede haber muchas cosas que podrían ser optimizadas, enfócate en lo que entregaría el mayor impacto. Podría ser una función que solo se llama una vez pero toma mucho tiempo para ejecutarse, o una función que se llama múltiples veces pero es relativamente rápida.
4. Antes de hacer cualquier cambio, asegúrate de tener una copia de seguridad de tu código original, así como una copia de los resultados actuales del perfilado. **También debes asegurar guardar (si es posible) el resultado del código.**
5. Vuelve a ejecutar el perfilador y compara el rendimiento. Si no se observan cambios, entonces regresa al paso 2. **Asegúrate de que la nueva versión del código mantenga la misma funcionalidad que el original (verifica los resultados).**

1.2 Perfilado de código en R

En el lenguaje de programación R, la herramienta de perfilado más utilizada viene con el paquete `profvis`. El paquete proporciona un envoltorio de la función `Rprof`, que es una función incorporada de R para perfilar código. El paquete `profvis` hace más fácil visualizar los resultados del perfilado en una interfaz web.

Para usar `profvis`, primero necesitas instalarlo desde CRAN:

```
install.packages("profvis")
```

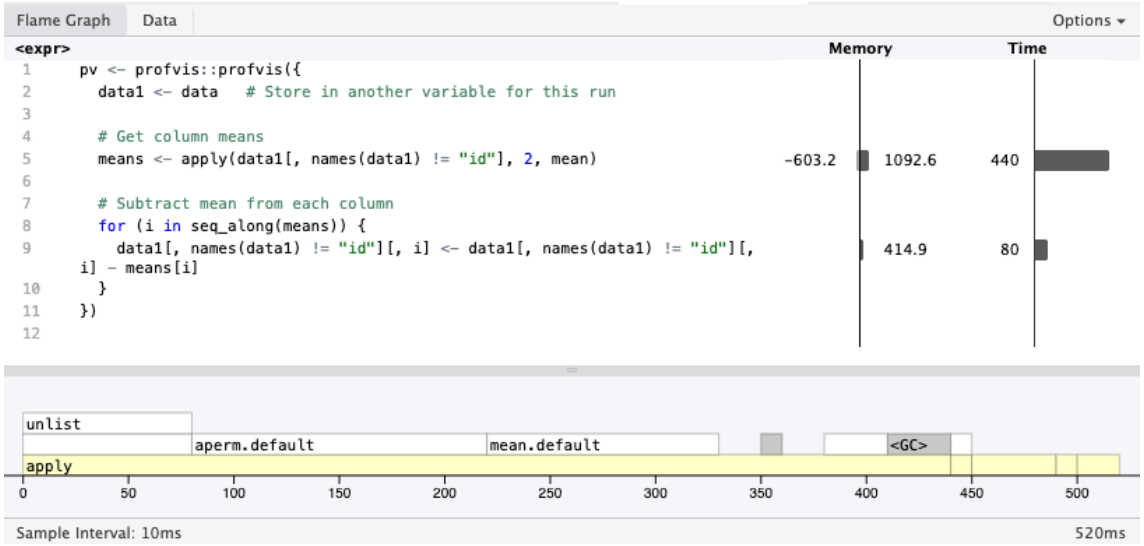
Luego, puedes usarlo para perfilar tu código R de esta manera:

```
library(profvis)

profvis({
  # Tu código R aquí
})
```

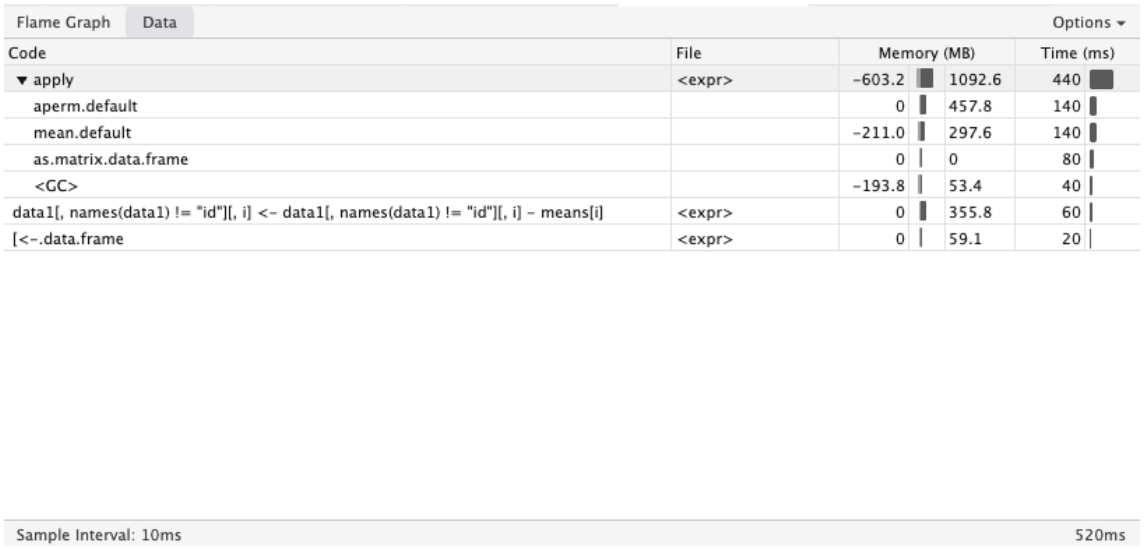
Esto ejecutará el código y generará una visualización de los resultados del perfilado en una nueva ventana del navegador. También puedes guardar la salida usando el paquete `htmlwidgets`:

```
pv <- profvis({
  # Tu código R aquí
})

htmlwidgets::saveWidget(pv, "profvis.html")
```

Una vez abierto, verás dos visualizaciones: el gráfico de llama y los datos. El gráfico de llama es una de las visualizaciones más útiles. Mapea directamente el tiempo y la memoria utilizados por cada línea de código



La visualización de datos muestra la distribución del tiempo gastado en cada función. Usando una estructura de árbol, que permite profundizar en la pila de llamadas.



> **Tip**
>
> Cuando desarrolles paquetes de R, es una buena idea emparejar tu llamada `profvis::profvis` con `devtools::load_all()` para asegurar que todo el código fuente esté disponible para el perfilador. De lo contrario, el gráfico de llama no mostrará tu código (dirá "no disponible").

1.3 Ejercicio: Identificando el cuello de botella[4]

```
# Generate data
times <- 4e5
cols <- 150
data <- as.data.frame(x = matrix(rnorm(times * cols, mean = 5), ncol = cols))
data <- cbind(id = paste0("g", seq_len(times)), data)

pv <- profvis::profvis({
  data1 <- data   # Store in another variable for this run
```

[4]Código copiado literalmente del paquete R `profvis` aquí.

```
  # Get column means
  means <- apply(data1[, names(data1) != "id"], 2, mean)

  # Subtract mean from each column
  for (i in seq_along(means)) {
    data1[, names(data1) != "id"][, i] <- data1[, names(data1) != "id"][, i] - means[i]
  }
})

htmlwidgets::saveWidget(pv, "profvis-slow-code.html")

# In interactive mode, we can directly view the profiling results
if (interactive())
  print(pv)
```

¿Puedes identificar dónde está el cuello de botella en el código? ¿Qué harías para acelerarlo?

2. Escribiendo Código Eficiente

⚠ Nota de Traducción

Esta versión del capítulo fue traducida de manera automática utilizando IA. El capítulo aún no ha sido revisado por un humano.

- El código R puede ser muy eficiente para tareas típicas, pero, a medida que el código comienza a aumentar en complejidad, es fácil que se vuelva ineficiente.
- Algunos consejos rápidos de R para código de computación eficiente:
  - Usa operaciones vectorizadas en lugar de bucles.
  - Trata de usar caché para evitar cálculos repetidos. ¡El caché también se puede hacer fuera de la memoria!
  - Evita pasos/procesamiento de datos innecesarios.
  - Reduce el número de operaciones de copia.

2.1 Operaciones Vectorizadas

La vectorización puede significar muchas cosas en programación, pero en R, la vectorización se refiere a usar funciones sobre vectores. Por ejemplo, en lugar de usar un bucle para sumar dos vectores juntos, puedes usar el operador `+` directamente en los vectores:

```
# Using a loop
set.seed(331)
a <- runif(1e3)
b <- runif(1e3)
result <- numeric(length(a))
for (i in seq_along(a)) {
  result[i] <- a[i] + b[i]
}

# Using vectorized operation
result <- a + b
```

Incluso podemos hacer benchmark del rendimiento de estos dos enfoques:

```
library(microbenchmark)
microbenchmark(
  loop = {
    result <- numeric(length(a))
    for (i in seq_along(a)) {
      result[i] <- a[i] + b[i]
    }
  },
  vectorized = {
    result <- a + b
  },
  unit = "relative"
)
```

```
Unit: relative
       expr      min       lq     mean   median       uq      max neval
```

```
      loop 2507.589 1396.289 785.1118 846.5277 653.7051 617.1289   100
vectorized    1.000    1.000   1.0000   1.0000   1.0000   1.0000   100
```

> Tip
>
> Los bucles for no siempre son malos. El problema principal está con el código dentro del bucle for. Si el código ya está vectorizado, entonces no hay necesidad de eliminar el bucle for (a menos que puedas vectorizar el bucle for mismo).

2.2 Cacheando cálculos

Muchas veces, es útil cachear cálculos que son costosos de computar. Por ejemplo, si tienes una función que toma mucho tiempo para ejecutarse, puedes almacenar el resultado en una variable y reutilizarlo más tarde en lugar de recalcularlo.

Aquí hay un mal ejemplo usando la secuencia de Fibonacci:

```
fibonacci <- function(n) {
  if (n <= 1) {
    return(n)
  }
  return(fibonacci(n - 1) + fibonacci(n - 2))
}

fibonacci_cached <- function(n) {
  prev <- numeric(n + 1)
  for (i in seq_len(n)) {
    if (i <= 1) {
      prev[i + 1] <- i
    } else {
      prev[i + 1] <- prev[i] + prev[i - 1]
    }
  }

  return(prev[n + 1])
}
```

Ambas funciones deberían devolver el mismo resultado, pero la segunda es significativamente más rápida ya que evita llamar la función recursivamente:

```
microbenchmark(
  fibonacci(10),
  fibonacci_cached(10),
  times = 10,
  unit = "relative",
  check = "equal"
)
```

```
Unit: relative
                expr      min       lq     mean   median       uq      max
        fibonacci(10) 23.26166 22.77055 17.36273 22.59429 19.82186 6.522117
 fibonacci_cached(10)  1.00000  1.00000  1.00000  1.00000  1.00000 1.000000
 neval
    10
    10
```

2.3 Cacheando cálculos (bis)

En el caso de cálculos grandes, también podemos guardar resultados en el disco. Por ejemplo, si estamos ejecutando una simulación/cálculo, uno por ciudad/escenario, podemos guardar los resultados en un archivo y leerlos más tarde. Aquí está cómo hacerlo:

Para cada valor de `i`, haz lo siguiente:

1. Verifica si el archivo `result_i.rds` existe.
2. Si no existe, ejecuta el cálculo y guarda el resultado en `result_i.rds`.
3. Si existe, lee el resultado de `result_i.rds`.

¡Así de simple! Aquí hay un ejemplo usando código R:

```r
# A complicated simulation function
simulate <- function(i, seed) {
  set.seed(seed)
  rnorm(1e5)
}

# Generating seeds for each iteration
set.seed(331)
nsims <- 100
seeds <- sample.int(.Machine$integer.max, nsims)

# Just for this example, we will use a tempfile
res_0 <- vector("list", length = nsims)
for (i in seq_len(nsims)) {

  # Creating the filename
  fn <- file.path(tempdir(), paste0(i, ".rds"))

  # Does the result already exist?
  if (file.exists(fn))
    res_0[[i]] <- readRDS(fn)
  else {
    # If not, run the simulation and save the result
    res_0[[i]] <- simulate(i, seed = i)
    saveRDS(res_0[[i]], fn)
  }

}
```

> **Tip**
>
> Cuando ejecutes simulaciones, es una buena práctica establecer semillas individuales para cada simulación (si estas son individualmente complejas). De esa manera, si el código falla, puedes volver a ejecutar solo las simulaciones fallidas sin tener que rehacer todas.

Además, es una buena idea envolver tu código en una llamada `tryCatch()` para manejar errores con gracia. De esta manera, si una simulación falla, puedes registrar el error y continuar con la siguiente simulación sin detener todo el proceso.

```r
# Just for this example, we will use a tempfile
res_0 <- vector("list", length = nsims)
for (i in seq_len(nsims)) {

  # Creating the filename
  fn <- file.path(tempdir(), paste0(i, ".rds"))

  # Does the result already exist?
  res <- tryCatch({
```

```
    if (file.exists(fn))
      readRDS(fn)
    else {
      # If not, run the simulation and save the result
      ans_i <- simulate(i, seed = i)
      saveRDS(ans_i, fn)
      ans_i
    }
  }, error = function(e) e)

  if (inherits(res, "error")) {
    message("Simulation ", i, " failed: ", res$message)
    next  # Skip to the next iteration
  }

  # We still store it, even if it failed
  res_0[[i]] <- res

}
```

Tip

La función `saveRDS` en R usa el argumento `compress = TRUE` como predeterminado. Comprimir los datos para ahorrar espacio es generalmente una buena idea, pero no si necesitas leer datos rápidamente. Entonces, si el espacio no es una restricción, puedes establecer `compress = FALSE` cuando guardes el archivo RDS para acelerar el proceso de lectura.

2.4 Cacheando cálculos en una ShinyApp

Abajo hay un ejemplo de una figura de plotly que está pre-grabada para una aplicación shiny. La idea es que, si la figura no necesita ser reactiva, siempre puedes pre-computar los resultados y almacenarlos en un archivo, en este caso, como un archivo HTML:

```
library(shiny)
library(bslib)
library(plotly)

# Like we did with the simulations, we have a default filename
fn <- "plotly.html"

# Notice I'm adding the www because, outside of the
# server call, this writes directly to the top level.
# Once reading, it will read from www.
if (!file.exists(file.path("www", fn))) {

  message("Creating the file...")

  # if it doesn't exist, then it creates it and saves it
  p <- plot_ly(x = 1:10, y = 1:10) %>% add_lines()
  htmlwidgets::saveWidget(
    p,
    file = "www/plotly.html",
    selfcontained = TRUE
    )
} else {
  message("The file already exists!")
}
```

```
# Define UI for app that draws a histogram ----
ui <- page_sidebar(
  # App title ----
  title = "Hello Shiny!",
  # Sidebar panel for inputs ----
  sidebar = sidebar(
    # Input: Slider for the number of bins ----
    sliderInput(
      inputId = "bins",
      label = "Number of bins:",
      min = 1,
      max = 50,
      value = 30
    )
  ),
  htmlOutput(outputId = "plotlyOutput")
)

server <- function(input, output) {

  output$plotlyOutput <- renderUI({
    tags$iframe(
      src = "plotly.html"
    )
  })

}

shinyApp(ui = ui, server = server)
```

2.5 Evitando pasos innecesarios

Muchas veces, podemos encontrar atajos para reducir la cantidad de procesamiento de datos que necesitamos hacer. Un gran ejemplo está en la función de regresión lineal `lm()`. La función `lm()` irá más allá de encontrar los coeficientes en un modelo lineal, también calculará residuos, valores ajustados, y más. En su lugar, podemos usar la función `lm.fit()` que solo calcula los coeficientes:

```
set.seed(331)
x <- rnorm(2e3)
y <- 2 + 3 * x + rnorm(2e3)

# Comparing
microbenchmark(
  lm = coef(lm(y ~ x)),
  lm_fit = coef(lm.fit(cbind(1, x), y)),
  times = 10,
  unit = "relative"
)
```

```
Unit: relative
   expr      min       lq     mean   median       uq      max neval
     lm 6.544506 6.534748 7.520741 6.256343 6.528294 15.80171    10
 lm_fit 1.000000 1.000000 1.000000 1.000000 1.000000  1.00000    10
```

2.6 Reduciendo operaciones de copia

Como en cualquier lenguaje de programación, las operaciones de copia en R pueden ser costosas. Más allá de aumentar la cantidad de memoria utilizada, las operaciones de copia requieren tiempo para asignar memoria y luego copiar los datos. R moderno minimiza estas usando copy-on-modify. Esto significa que R no copiará un objeto hasta que sea modificado. Por ejemplo, el siguiente código hace múltiples copias de `X`, pero es hasta la última línea que R realmente hace una copia de `X`:

```
set.seed(331)
X <- runif(1e4)
Y <- X
Z <- X

# Checking the address of the objects
library(lobstr)
obj_addr(X)
## [1] "0x5604ba715130"
obj_addr(Y)
## [1] "0x5604ba715130"
obj_addr(Z)
## [1] "0x5604ba715130"
```

Modificar `X` disparará una operación de copia, y las direcciones de `Y` y `Z` permanecerán iguales, mientras que `X` tendrá una nueva dirección:

```
# Modifying X
X[1] <- 100  # This is when R makes a copy of X
obj_addr(X)
## [1] "0x5604badf9e50"
obj_addr(Y)
## [1] "0x5604ba715130"
obj_addr(Z)
## [1] "0x5604ba715130"
```

II Computación paralela

3. Introducción a la Computación Paralela

⚠ Nota de Traducción

Esta versión del capítulo fue traducida de manera automática utilizando IA. El capítulo aún no ha sido revisado por un humano.

4. Introducción

Aunque la mayoría de la gente ve R como un lenguaje de programación lento, tiene características poderosas que aceleran dramáticamente tu código[5]. Aunque R no fue necesariamente construido para la velocidad, hay algunas herramientas y formas en las que podemos acelerar R. Este capítulo introduce lo que entenderemos como computación de alto rendimiento en R.

[5]No obstante, esta afirmación se puede decir sobre casi cualquier lenguaje de programación; hay ejemplos notables como el paquete R `data.table` [2] que ha sido demostrado que supera a la mayoría de herramientas de manipulación de datos.

5. Computación de Alto Rendimiento: Una visión general

Desde la perspectiva de R, podemos pensar en HPC en términos de dos o tres cosas:[6] Big data, computación paralela, y código compilado.

5.1 Big Data

Cuando hablamos de big data, nos referimos a casos donde tu computadora lucha para manejar un conjunto de datos. Un ejemplo típico de esto último es cuando el número de observaciones (filas) en tu marco de datos es demasiado para ajustar un modelo de regresión lineal. En lugar de comprar una computadora más grande, hay muchas buenas soluciones para resolver problemas relacionados con la memoria:

- **Almacenamiento fuera de memoria**. La idea es simple, en lugar de usar tu RAM para cargar los datos, usa otros métodos para cargar los datos. Dos alternativas notables son los paquetes R bigmemory e implyr. El paquete `bigmemory` proporciona métodos para usar matrices "*respaldadas por archivo*". Por otro lado, `implyr` implementa un envoltorio para acceder a Apache Impala, un motor de consultas SQL para un clúster ejecutando Apache Hadoop.
- **Algoritmos eficientes para big data**: Para evitar quedarte sin memoria con tu análisis de regresión, los paquetes R biglm y biglasso entregan alternativas altamente eficientes a `glm` y `glmnet`, respectivamente. Ahora, si tus datos caben en tu RAM, pero aún luchas con la manipulación de datos, el paquete data.table es la solución.
- **Almacénalo más eficientemente**: Finalmente, cuando se trata de álgebra lineal, el paquete R Matrix brilla con sus clases formales y métodos para manejar Matrices Dispersas, *es decir*, matrices grandes cuyas entradas son principalmente ceros; por ejemplo, los objetos `dgCMatrix`. Además, `Matrix` viene incluido con R, lo que lo hace aún más atractivo.

[6] Asegúrate de revisar CRAN Task View on HPC.

5.2 Computación paralela

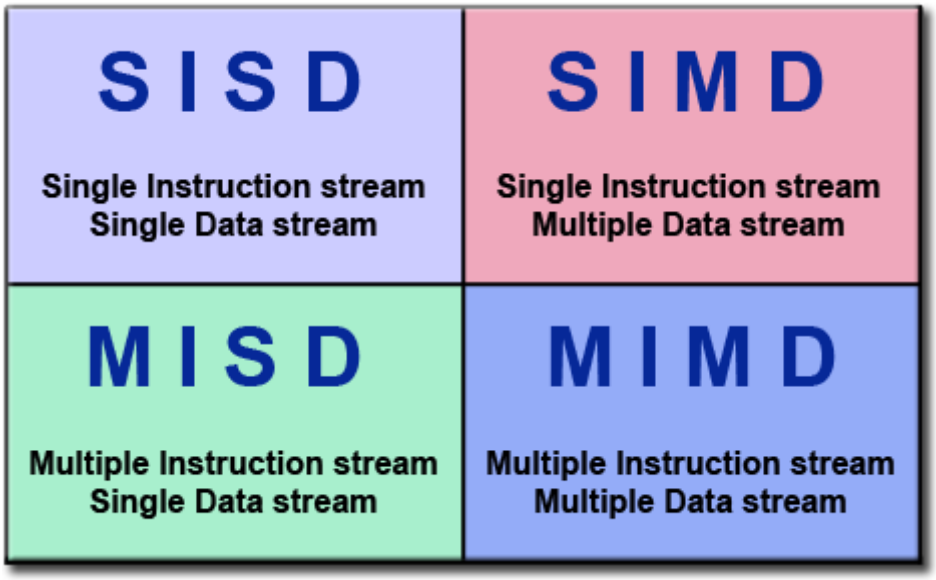


Figure 5.1: Taxonomía Clásica de Flynn (Blaise Barney, **Introduction to Parallel Computing**, Lawrence Livermore National Laboratory)

Nos enfocaremos en el **S**ingle **I**nstruction stream **M**ultiple **D**ata stream (Flujo de Instrucción Única Flujo de Datos Múltiple).

En términos generales, un programa de computación paralela es uno en el cual usamos dos o más *hilos computacionales* simultáneamente. Aunque hilo computacional usualmente significa núcleo, hay múltiples niveles en los que un programa de computadora puede ser paralelizado. Para entender esto, primero necesitamos ver qué compone una computadora moderna:

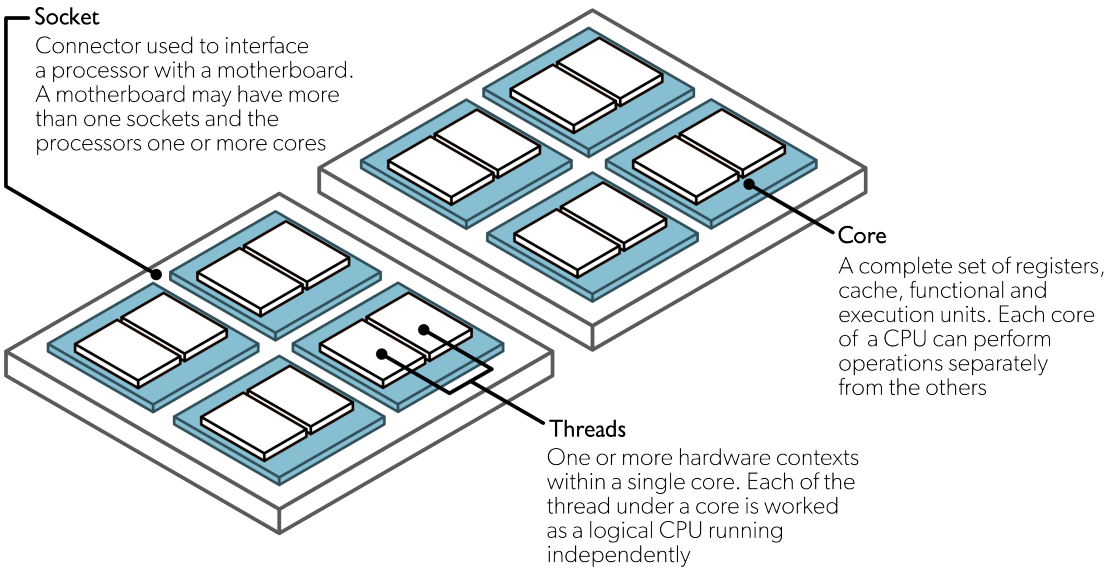
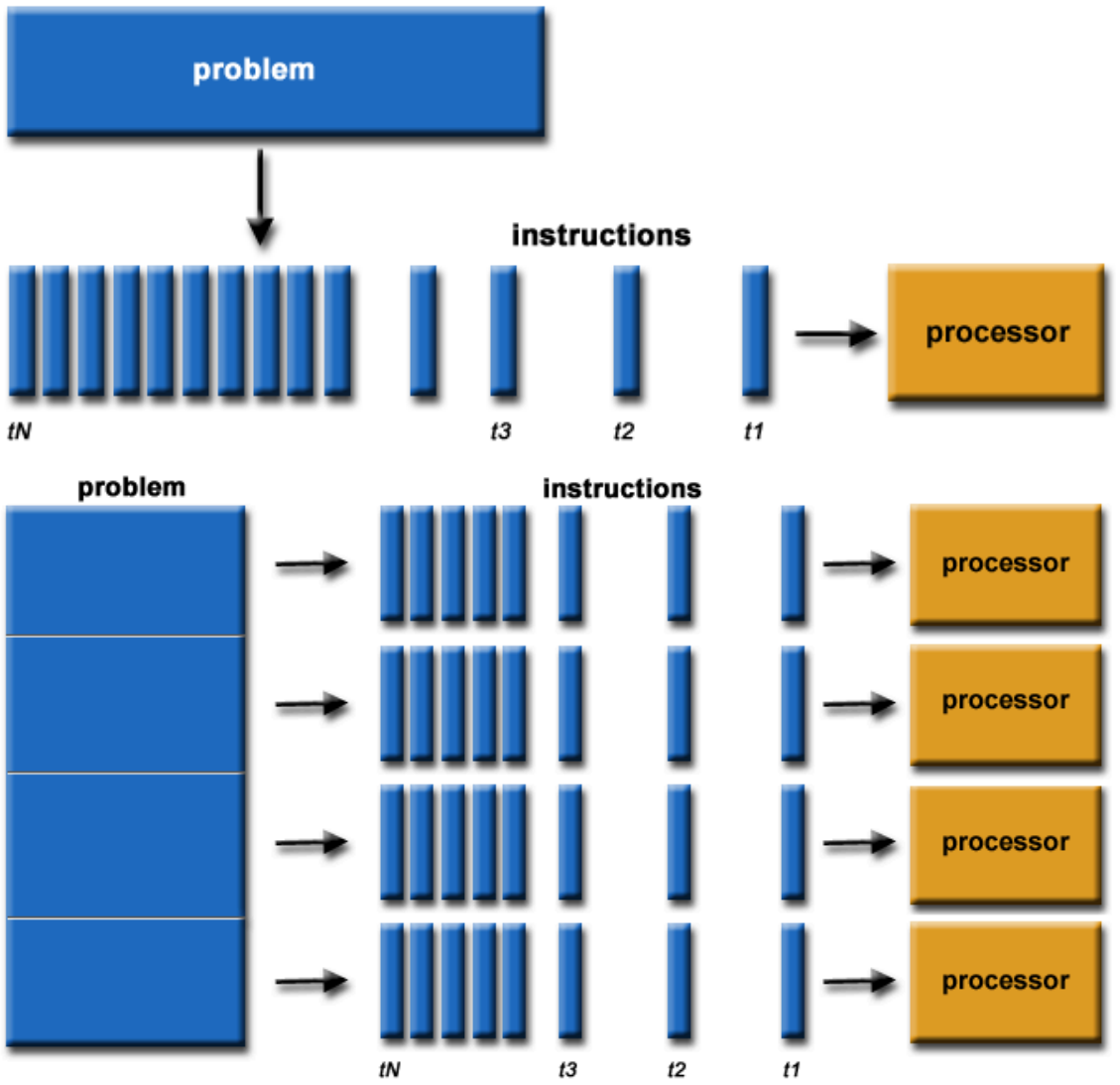


Figure 5.2: Fuente: Figura original de la documentación del consorcio LUMI [3]

Extensiones SIMD de Flujo [SSE] y Extensiones de Vector Avanzado [AVX]

5.2.1 Serial vs. Paralelo



Fuente: Blaise Barney, **Introduction to Parallel Computing**, Lawrence Livermore National Laboratory

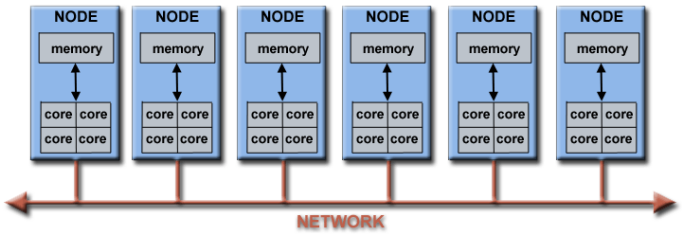


Figure 5.3: fuente: Blaise Barney, **Introduction to Parallel Computing**, Lawrence Livermore National Laboratory

5.3 Computación de alto rendimiento en R

5.3.1 Algo de vocabulario para HPC

En términos crudos

- Supercomputadora: Una **sola** máquina grande con miles de núcleos/GPGPUs.

- Computación de Alto Rendimiento (HPC): **Múltiples** máquinas dentro de una **sola** red.
- Computación de Alto Rendimiento (HTC): **Múltiples** máquinas a través de **múltiples** redes.

Puede que no tengas acceso a una supercomputadora, pero ciertamente, los clústeres HPC/HTC son más accesibles hoy en día, *p. ej.*, AWS proporciona un servicio para crear clústeres HPC a bajo costo (supuestamente, ya que nadie entiende cómo funciona la tarificación)

5.4 GPU vs. CPU

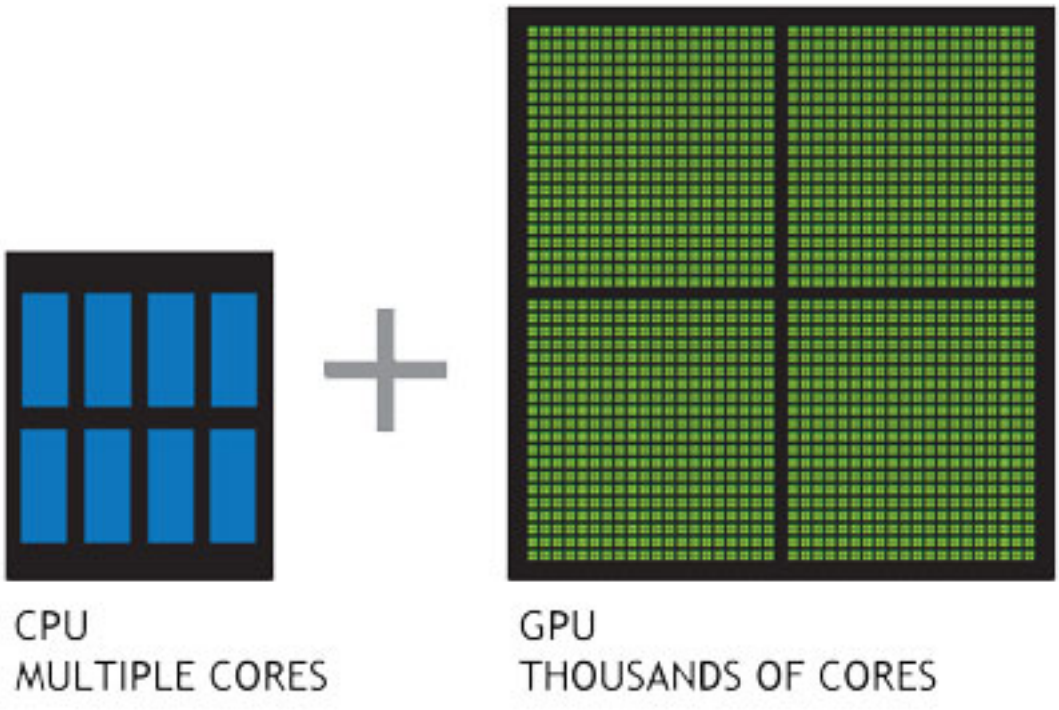


Figure 5.4: NVIDIA Blog

- ¿Por qué usar OpenMP si GPU está *adecuado para operaciones intensivas en cómputo*? Bueno, principalmente porque OpenMP es **MUY** fácil de implementar (más fácil que CUDA, que es la forma más fácil de usar GPU).[7]

[7] Laboratorios Nacionales Sandia comenzó el proyecto Kokkos, que proporciona una biblioteca C++ que se ajusta a todo para programación paralela. Más información en el sitio wiki de Kokkos.

5.5 ¿Cuándo es una buena idea?

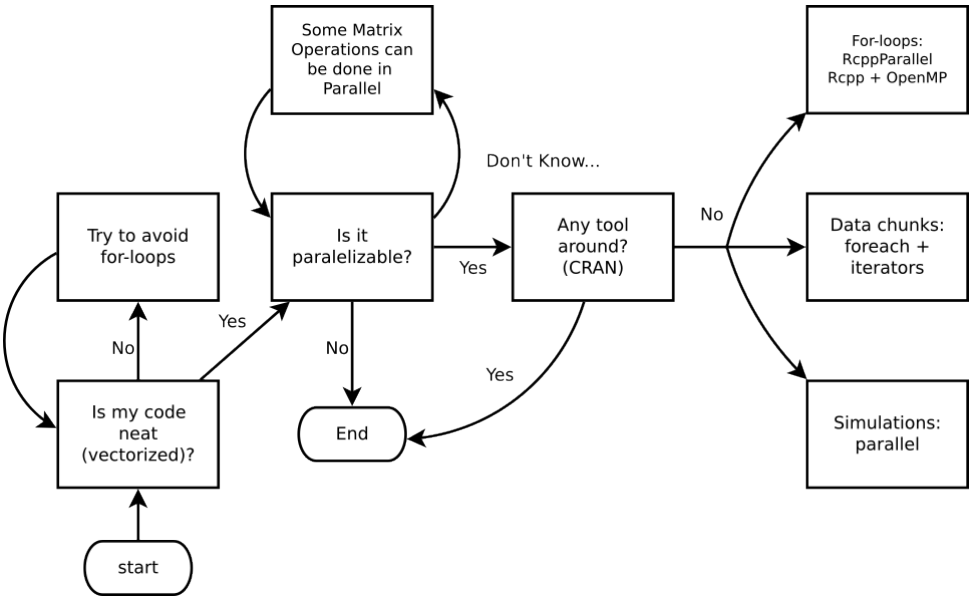


Figure 5.5: ¡Hazte estas preguntas antes de saltar a HPC!

5.6 Computación paralela en R

Mientras hay varias alternativas (solo echa un vistazo a la Vista de Tareas de Computación de Alto Rendimiento), nos enfocaremos en los siguientes paquetes R para **paralelismo explícito**:

- **parallel**: Paquete R que proporciona '[s]oporte para computación paralela, incluyendo generación de números aleatorios'.
- **future**: '[U]na API Future ligera y unificada para procesamiento secuencial y paralelo de expresiones R via futures.'
- **Rcpp** + OpenMP: Rcpp es un paquete R para integrar R con C++ y OpenMP es una biblioteca para paralelismo de alto nivel para C/C++ y FORTRAN.

Otros pero no usados aquí

- **foreach** para iterar a través de listas en paralelo.
- **Rmpi** para crear clústeres MPI.

Y herramientas para paralelismo implícito (herramientas listas para usar que permiten al programador no preocuparse por la paralelización):

- **gpuR** para manipulación de Matrices usando GPU
- **tensorflow** una interfaz R a TensorFlow.

Una tonelada de otros tipos de recursos, notablemente las herramientas para trabajar con programadores de lotes como Slurm, y HTCondor.

6. El Paquete parallel

⚠ Nota de Traducción

Esta versión del capítulo fue traducida de manera automática utilizando IA. El capítulo aún no ha sido revisado por un humano.

Aunque R no fue construido para computación paralela, existen múltiples formas de paralelizar tu código R. Una de estas es el paquete `parallel`. Este paquete de R, incluido con R base, proporciona varias funciones para paralelizar código R usando computación embarazosamente paralela, es decir, una estrategia de tipo divide-y-vencerás. La idea básica es iniciar múltiples sesiones de R (usualmente llamadas procesos hijos), conectar la sesión principal con estas, y enviarles instrucciones. Esta sección cubre un flujo de trabajo común para trabajar con el parallel de R.

6.1 Flujo de trabajo paralelo

(Usualmente) Hacemos lo siguiente:

1. Crear un cluster `PSOCK/FORK` (u otro) usando `makePSOCKCluster/makeForkCluster` (o `makeCluster`). Cuántos procesos hijos dependeŕa de cuántos núcleos tiene tu computadora. Una regla general es usar `parallel::detectCores() - 1` núcleos (para dejar uno libre para el resto de tu computadora).
2. Copiar/preparar cada sesión de R (si estás usando un cluster `PSOCK`):
   a. Copiar objetos con `clusterExport`. Estos serían todos los objetos que necesitas en las sesiones hijas.
   b. Pasar expresiones con `clusterEvalQ`. Esto incluiría cargar paquetes de R y otro código en las otras sesiones.
   c. Establecer una semilla (si estás haciendo algo que involucra aleatoriedad)
3. Hacer tu llamada: `parApply`, `parLapply`, etc.
4. Detener el cluster con `clusterStop`

Como mencionamos más adelante, el paso 2 dependerá del tipo de cluster que estés usando. Si estás usando una conexión Socket (`PSOCK` cluster), entonces las sesiones de R generadas serán completamente nuevas (sin datos o paquetes de R precargados); mientras que usar una conexión Fork (`FORK` cluster) copiará la sesión actual de R, incluyendo todos los objetos y paquetes cargados.

6.2 Tipos de clusters: PSOCK

- Se puede crear con `makePSOCKCluster`
- Crea sesiones de R completamente nuevas (por lo que nada se hereda del maestro), ej.

```
# Esto crea un cluster con 4 sesiones de R
cl <- makePSOCKCluster(4)
```

- Las sesiones hijas están conectadas a la sesión maestra a través de conexiones Socket
- Se puede crear fuera de la computadora actual, **es decir**, ¡entre múltiples computadoras!

6.3 Tipos de clusters: Fork

- Fork Cluster `makeForkCluster`:
- Usa Forking del SO,

- Copia la sesión actual de R localmente (por lo que todo se hereda del maestro hasta ese punto).
- Los datos solo se duplican si se alteran (¡necesito verificar cuándo sucede esto!)
- No disponible en Windows.

Otros tipos están disponibles a través de la función `makeCluster` del paquete de R **snow** (Simple Network of Workstations). Estos incluyen clusters MPI (Message Passing Interface) y clusters Slurm (Socket).

6.4 Un programa plantilla

El siguiente bloque de código muestra una plantilla para usar el paquete `parallel` en R. Puedes copiar esto y comentar las partes que no necesites:

```
library(parallel)

# 1. CREAR UN CLUSTER ----------------
nnodes <- 4L # ¡Podría ser menos o más!
cl <- makePSOCKcluster(nnodes)

# 2. PREPARAR EL CLUSTER -------------

# Principalmente si se usa PSOCK
clusterEvalQ(cl, {
  library(...) # Cargar los paquetes necesarios
  source(...) # Cargar scripts adicionales
})

# Siempre si estás generando números aleatorios
clusterSetRNGStream(cl, 123)

# 3. HACER TU LLAMADA ----------------------
ans <- parLapply(
  cl,
  ... lista larga para iterar ...,
  function(x) {
    ...
  },
  ... argumentos adicionales ...
  )

# 4. DETENER EL CLUSTER
stopCluster(cl)
```

Generalmente, la `... lista larga para iterar ...` será un vector u otra lista que contenga datos (ej., conjuntos de datos individuales), una secuencia de números (ej., del 1 al 1000), una lista de rutas de archivos (si estuvieras procesando archivos individualmente), o directamente una secuencia corta con números del 1 al número de nodos (aplicación menos común).

Cuando llamas `parLapply` o `parSapply` (las versiones paralelas de `lapply` y `sapply` respectivamente), la llamada a la función dividirá automáticamente las iteraciones entre nodos usando la función `splitIndices`. Aquí hay un ejemplo de lo que sucede bajo el capó:

```
# Distribuyendo 9 iteraciones entre dos núcleos
(n_iterations <- parallel::splitIndices(nx = 9, ncl = 2))
```

```
[[1]]
[1] 1 2 3 4

[[2]]
[1] 5 6 7 8 9
```

Lo que significa que la primera sesión de R obtendrá 4 trabajos, mientras que la segunda sesión de R obtendrá 5 trabajos. De esta manera, cada sesión de R generada (sesión hija) obtiene un número similar de iteraciones.

6.5 Ejemplo: Reemplazar un for-loop con `parLapply`

Uno de los casos de uso más comunes de la computación paralela es reemplazar un for-loop con una versión paralela. En este ejemplo, llamaremos a `runif(10)` 1,000 veces, comparando un simple for-loop con `parLapply`.

6.5.1 Versión serial usando un for-loop

La versión serial itera sobre las simulaciones usando un for-loop:

```
nsims <- 1000

# Primero, crear semillas para las ejecuciones individuales
set.seed(1231)
seeds <- sample.int(1e7, nsims)

# Versión serial usando un for-loop
result_serial <- vector("list", nsims)
for (i in seq_len(nsims)) {
  set.seed(seeds[i])
  result_serial[[i]] <- runif(10)
}

# Viendo los primeros resultados
head(result_serial, 2)
```

```
[[1]]
 [1] 0.38201608 0.30000010 0.03890397 0.51085385 0.63979380 0.26806211
 [7] 0.86383563 0.64311030 0.17286217 0.89551119

[[2]]
 [1] 0.23208922 0.65968122 0.55613334 0.75545780 0.25803644 0.89517280
 [7] 0.90952159 0.69711625 0.02608031 0.60768982
```

El lector puede notar que pre-generamos un vector con semillas para cada simulación. Si bien esta no es una práctica común en computación serial, como veremos, es una estrategia común para asegurar la reproducibilidad en computación paralela.

6.5.2 Versión paralela usando `parLapply`

Podemos reemplazar el for-loop con `parLapply` siguiendo la plantilla que describimos anteriormente. Nótese la estrategia de semillas: en lugar de usar `clusterSetRNGStream`, estamos pre-generando un vector de semillas y estableciendo la semilla dentro de cada simulación. Esto hace que los resultados sean 100% reproducibles independientemente del número de núcleos/hilos utilizados.

```
library(parallel)

# 1. CREAR UN CLUSTER ----------------
cl <- makePSOCKcluster(4L)

# 2. PREPARAR EL CLUSTER -------------
# Exportamos el vector de semillas a todos los workers
clusterExport(cl, "seeds")

# 3. HACER TU LLAMADA ----------------------
result_parallel <- parLapply(cl, seq_len(nsims), \(i) {
```

```
  # Esto lo hace 100% reproducible, independientemente del
  # número de núcleos/hilos que estemos usando
  set.seed(seeds[i])
  runif(10)
})

# 4. DETENER EL CLUSTER
stopCluster(cl)

# Los resultados son idénticos
all.equal(result_serial, result_parallel)
```

```
[1] TRUE
```

i Semillas por simulación vs `clusterSetRNGStream`

El enfoque utilizado aquí—generar semillas de antemano y llamar a `set.seed()` dentro de cada simulación—es una alternativa a `clusterSetRNGStream`. Las diferencias clave son:

- **`clusterSetRNGStream`** inicializa el generador de números aleatorios L'Ecuyer-CMRG a través del cluster. Los flujos dependen del número de workers, por lo que los resultados pueden cambiar si cambias el número de núcleos.
- **Semillas por simulación** (como se muestra aquí) asigna una semilla única a cada iteración. Dado que la semilla está ligada al índice de la simulación y no al worker, los resultados son idénticos sin importar cuántos núcleos se utilicen.

Ambos enfoques son válidos. Las semillas por simulación son especialmente útiles cuando quieres resultados que sean invariantes al número de hilos, por ejemplo, al depurar o comparar ejecuciones entre diferentes máquinas. Nótese que combinar ambas estrategias es generalmente redundante: una vez que llamas a `set.seed()` dentro de cada iteración, el flujo RNG del worker inicializado por `clusterSetRNGStream` ya no afecta los números generados en esa iteración. Usar `clusterSetRNGStream` junto con semillas por simulación solo importaría si también dependes de RNG fuera de la llamada `set.seed()` por iteración, por ejemplo, en código de configuración ejecutado en los workers.

6.6 Ejemplo: Ejecutar una regresión lineal a través de múltiples columnas

En genómica, es común analizar datos genómicos a nivel de genes comparando niveles de expresión contra algún fenotipo/enfermedad. Un análisis simple consiste en ejecutar una regresión lineal a través de múltiples columnas (genes) de un data frame. El siguiente bloque de código genera algunos datos artificiales que podemos usar para este ejemplo:

```
set.seed(331)
n_genes <- 10000
n_obs <- 1000

# Una matriz aleatoria de ómicas
X_genes <- rnorm(n_obs * n_genes) |>
  matrix(nrow = n_obs)

# Un fenotipo aleatorio (completamente no relacionado para este ejemplo)
Y <- rnorm(n_obs) |> cbind()
```

Envolveremos el análisis en una función para poder hacer benchmarking. Usaremos la función `lapply` para iterar sobre las columnas de `X_genes`

```r
ols_serial <- function(X, Y) {
  lapply(
    X = seq_len(n_genes),
    FUN = \(i) {lm.fit(X[, i, drop = FALSE], Y) |> coef()}
  ) |> do.call(what = rbind)
}

# Llamando la función y viendo las primeras filas
ols_serial(X_genes, Y) |> head()
```

```
               x1
[1,]  0.029403088
[2,]  0.008907854
[3,] -0.027246099
[4,] -0.031280262
[5,] -0.001309752
[6,]  0.066971469
```

> **Tip**
>
> Como hicimos en la sección de programación eficiente, en lugar de usar `lm()` o `glm()`, podemos usar `lm.fit()` para mejor rendimiento. La función `lm.fit()` hace menos que la función `lm()` al omitir el cálculo de residuos y otras sobrecargas, haciéndola más rápida para conjuntos de datos grandes.

Usando computación paralela (y siguiendo la plantilla que presentamos anteriormente), esto podría hacerse de la siguiente manera con el paquete parallel:

```r
library(parallel)

ols_parallel <- function(X, Y, ncores) {
  # 1. CREAR UN CLUSTER ----------------
  cl <- makePSOCKcluster(ncores)

  # Esto será llamado al salir de la función
  on.exit(stopCluster(cl))

  # 2. PREPARAR EL CLUSTER -------------
  # Copiamos los datos
  clusterExport(cl, c("X", "Y"), envir = environment())

  # 3. HACER TU LLAMADA ----------------------
  parLapply(
    cl,
    seq_len(n_genes),
    function(i) {
      lm.fit(X[, i, drop = FALSE], Y) |> coef()
    }
  ) |> do.call(what = rbind)
}

# Verificando que funciona
ols_parallel(X_genes, Y, ncores = 4L) |> head()
```

```
               x1
[1,]  0.029403088
[2,]  0.008907854
[3,] -0.027246099
[4,] -0.031280262
```

```
[5,] -0.001309752
[6,]  0.066971469
```

> Tip
>
> Al igual que `return()`, `on.exit()` solo puede usarse dentro de una llamada a función. Podríamos haber usado `stopCluster(cl)` al final como hacemos en nuestro ejemplo de plantilla, pero el beneficio de usar `on.exit()` es que será llamado automáticamente cuando la función termine, incluso si ocurre un error. Esto ayuda a asegurar que el cluster siempre se detenga apropiadamente.

Ahora que tenemos la función implementada, podemos proceder a (1) comparar resultados y (2) medir rendimiento.

```
library(microbenchmark)

microbenchmark(
  serial = ols_serial(X_genes, Y),
  parallel = ols_parallel(X_genes, Y, ncores = 4L),
  times = 10L,
  check = "identical"
)
```

```
Unit: milliseconds
     expr       min        lq      mean    median        uq       max neval
   serial  622.6059  641.6958  663.7625  670.6183  676.8962  721.2354    10
 parallel 1838.0423 1846.0771 1872.6988 1876.7738 1891.0929 1913.3443    10
```

De la comparación, podemos ver que la versión paralela es significativamente más lenta que la versión serial. Dos cosas a notar aquí son (a) la tarea que estamos ejecutando ya es rápida (alrededor de 0.3 segundos en promedio para la ejecución serial) y (b) hay un costo de sobrecarga asociado con crear, preparar y detener el cluster. Como mencionamos anteriormente, las optimizaciones paralelas solo tienen sentido si tu código ya está tomando una cantidad significativa de tiempo, haciendo que el costo de sobrecarga asociado con la configuración sea relativamente pequeño. La siguiente implementación de la función debería hacerla significativamente más rápida:

```
ols_parallel2 <- function(cl) {
  # 1. CREAR UN CLUSTER ----------------
  # 2. PREPARAR EL CLUSTER -------------
  # Ya no es necesario ya que estamos manejando el núcleo fuera

  # 3. HACER TU LLAMADA ----------------------
  parLapply(
    cl,
    seq_len(n_genes),
    function(i) {
      lm.fit(X_genes[, i, drop = FALSE], Y) |> coef()
    }
  ) |> do.call(what = rbind)
}

# Verificando que funciona
cl <- makePSOCKcluster(4)
clusterExport(cl, c("X_genes", "Y"))
ols_parallel2(cl) |> head()
```

```
              x1
[1,]  0.029403088
[2,]  0.008907854
[3,] -0.027246099
```

```
[4,] -0.031280262
[5,] -0.001309752
[6,]  0.066971469
```

```
# 4. DETENER EL CLUSTER
stopCluster(cl)
```

Las principales diferencias de la versión anterior de la función son:

1. Estamos creando el cluster fuera de la función y pasándolo como argumento.
2. Estamos exportando las variables `X_genes` y `Y` al cluster solo una vez, lo cual también debería reducir significativamente la sobrecarga.
3. Debido al paso anterior, ahora estamos llamando `X_genes` directamente en la función principal.
4. El cluster se detiene fuera de la llamada a la función (ya que la función ya no maneja el objeto cluster).

Midamos el rendimiento para ver qué tan más rápida es la versión paralela.

```
library(microbenchmark)

# Necesitamos preparar el cluster de antemano
cl <- makePSOCKcluster(4)
clusterExport(cl, c("X_genes", "Y"))

microbenchmark(
  serial = ols_serial(X_genes, Y),
  parallel = ols_parallel2(cl),
  times = 10L,
  check = "identical"
)
```

```
Unit: milliseconds
     expr      min       lq     mean   median       uq      max neval
   serial 584.7565 626.5071 650.8558 649.9023 675.0508 712.9661    10
 parallel 285.3481 310.7722 322.2616 322.3116 331.1869 363.3274    10
```

```
# Necesitamos detener el cluster
stopCluster(cl)
```

Ahora, la versión paralela es significativamente más rápida que la versión serial. Solo usar el paquete parallel (o cualquier otro paquete que se pueda usar para computación paralela) no garantiza un rendimiento mejorado.

6.7 Más ejemplos

Los siguientes tres ejemplos son una aplicación simple del paquete en la cual estamos ejecutando explícitamente tantas réplicas como hilos tiene el cluster. Generalmente, el número de réplicas será una función de los datos.

6.7.1 Ej 1: RNG Paralelo con `makePSOCKCluster`

> **Caution**
>
> Usar más hilos que núcleos disponibles en tu computadora nunca es una buena idea. Como regla general, los clusters deberían crearse usando `parallel::detectCores() - 1` núcleos (para dejar uno libre para el resto de tu computadora.)

```
# 1. CREAR UN CLUSTER
library(parallel)
nnodes <- 4L
cl     <- makePSOCKcluster(nnodes)
# 2. PREPARAR EL CLUSTER
clusterSetRNGStream(cl, 123) # Equivalente a `set.seed(123)`
# 3. HACER TU LLAMADA
ans <- parSapply(cl, 1:nnodes, function(x) runif(1e3))
(ans0 <- var(ans))
```

```
              [,1]          [,2]          [,3]          [,4]
[1,]  0.0861888293 -0.0001633431  5.939143e-04 -3.672845e-04
[2,] -0.0001633431  0.0853841838  2.390790e-03 -1.462154e-04
[3,]  0.0005939143  0.0023907904  8.114219e-02 -4.714618e-06
[4,] -0.0003672845 -0.0001462154 -4.714618e-06  8.467722e-02
```

Asegurándonos de que es reproducible

```
# ¡Quiero obtener lo mismo!
clusterSetRNGStream(cl, 123)
ans1 <- var(parSapply(cl, 1:nnodes, function(x) runif(1e3)))
# 4. DETENER EL CLUSTER
stopCluster(cl)
all.equal(ans0, ans1) # ¡Todo igual!
```

```
[1] TRUE
```

6.7.2 Ej 2: RNG Paralelo con `makeForkCluster`

En el caso de `makeForkCluster`

```
# 1. CREAR UN CLUSTER
library(parallel)
# El fork cluster copiará el objeto -nsims-
nsims  <- 1e3
nnodes <- 4L
cl     <- makeForkCluster(nnodes)
# 2. PREPARAR EL CLUSTER
clusterSetRNGStream(cl, 123)
# 3. HACER TU LLAMADA
ans <- do.call(cbind, parLapply(cl, 1:nnodes, function(x) {
  runif(nsims) # ¡Mira! usamos el objeto nsims!
               # Esto habría fallado en makePSOCKCluster
               # si no copiamos -nsims- primero.
  }))
(ans0 <- var(ans))
```

```
              [,1]          [,2]          [,3]          [,4]
[1,]  0.0861888293 -0.0001633431  5.939143e-04 -3.672845e-04
[2,] -0.0001633431  0.0853841838  2.390790e-03 -1.462154e-04
[3,]  0.0005939143  0.0023907904  8.114219e-02 -4.714618e-06
[4,] -0.0003672845 -0.0001462154 -4.714618e-06  8.467722e-02
```

Nuevamente, queremos asegurarnos de que esto es reproducible

```
# Misma secuencia con misma semilla
clusterSetRNGStream(cl, 123)
ans1 <- var(do.call(cbind, parLapply(cl, 1:nnodes, function(x) runif(nsims))))
ans0 - ans1 # Una matriz de ceros
```

```
     [,1] [,2] [,3] [,4]
[1,]    0    0    0    0
[2,]    0    0    0    0
[3,]    0    0    0    0
[4,]    0    0    0    0
```

```
# 4. DETENER EL CLUSTER
stopCluster(cl)
```

Bueno, si eres usuario de Mac-OS/Linux, hay una forma más directa de hacer esto...

6.7.3 Ej 3: RNG Paralelo con `mclapply` (Forking sobre la marcha)

En el caso de `mclapply`, ¡el forking (creación del cluster) se hace sobre la marcha!

```
# 1. CREAR UN CLUSTER
library(parallel)
# El fork cluster copiará el objeto -nsims-
nsims  <- 1e3
nnodes <- 4L
# cl     <- makeForkCluster(nnodes) # mclapply lo hace sobre la marcha
# 2. PREPARAR EL CLUSTER
set.seed(123)
# 3. HACER TU LLAMADA
ans <- do.call(cbind, mclapply(1:nnodes, function(x) runif(nsims)))
(ans0 <- var(ans))
```

```
              [,1]         [,2]         [,3]         [,4]
[1,]  0.0835876419  0.002496611 -0.002543547 0.0008974346
[2,]  0.0024966111  0.084666939 -0.001166369 0.0039493749
[3,] -0.0025435465 -0.001166369  0.083853454 0.0020753217
[4,]  0.0008974346  0.003949375  0.002075322 0.0845621058
```

Una vez más, queremos asegurarnos de que esto es reproducible

```
# Misma secuencia con misma semilla
set.seed(123)
ans1 <- var(do.call(cbind, mclapply(1:nnodes, function(x) runif(nsims))))
ans0 - ans1 # Una matriz de ceros
```

```
              [,1]          [,2]          [,3]         [,4]
[1,] -5.626229e-05  0.0013659620 -0.0044675733 0.0040644556
[2,]  1.365962e-03  0.0038925145 -0.0024525109 0.0005280898
[3,] -4.467573e-03 -0.0024525109 -0.0029351885 0.0002855262
[4,]  4.064456e-03  0.0005280898  0.0002855262 0.0006822644
```

```
# 4. DETENER EL CLUSTER
# stopCluster(cl) ya no es necesario hacer esto
```

6.8 Ejercicio: Costos de sobrecarga

Compara el tiempo de tomar la suma de 100 números cuando está paralelizado versus no. Para la versión no paralelizada (serializada), usa lo siguiente:

```
set.seed(123)
x <- runif(n=100)

serial_sum <- function(x){
  x_sum <- sum(x)
```

```
  return(x_sum)
}
```

Para la versión paralelizada, sigue este esquema

```
library(parallel)
```

```
set.seed(123)
x <- runif(n=100)

parallel_sum <- function(){


  # Establecer número de núcleos a usar
  # hacer cluster y exportar al cluster la variable x
  # Usar "función de división para dividir x en tantos trozos como el número de núcleos

  # Calcular sumas parciales haciendo algo como:

  partial_sums <- parallel::parSapply(cl, x_split, sum)

  # Detener el cluster

  # Sumar y devolver las sumas parciales

}
```

Compara el tiempo de los dos enfoques:

```
microbenchmark::microbenchmark(
  serial   = serial_sum(x),
  parallel = parallel_sum(x),
  times    = 10,
  unit     = "relative"
)
```

III Trabajando con un Cluster

7. Fundamentos de SLURM

⚠ Nota de Traducción

Esta versión del capítulo fue traducida de manera automática utilizando IA. El capítulo aún no ha sido revisado por un humano.

8. ¿Qué es Slurm?

> **i** Note
>
> La mayor parte de esta sección fue extraída de la viñeta del paquete R `slurmR` "Working with Slurm."

Hoy en día, los clústeres de computación de alto rendimiento (HPC) son herramientas comúnmente disponibles tanto en configuraciones en la nube como fuera de ella. Slurm Work Manager (anteriormente *Simple Linux Utility for Resource Manager*) es un programa escrito en C que se usa para gestionar eficientemente recursos en clústeres HPC. El paquete R slurmR–que estaremos usando en este libro–proporciona herramientas para usar R en configuraciones HPC que funcionan con Slurm. Proporciona envoltorios y funciones que permiten al usuario integrar sin problemas su pipeline de análisis con clústeres HPC, enfatizando en proporcionar al usuario una familia de funciones similares a las que proporciona el paquete R parallel.

8.1 Definiciones

Primero, algunos puntos de discusión importantes dentro del contexto de Slurm+R que los usuarios, en general, encontrarán útiles. La mayoría de los puntos tienen que ver con opciones disponibles para Slurm, y en particular, con el comando `sbatch` que se usa para enviar trabajos en lote a Slurm. Los usuarios que han usado Slurm en el pasado pueden desear omitir esto y continuar leyendo la siguiente sección.

- **Nodo** Una sola computadora en el HPC: Muchas veces los trabajos se enviarán a un solo nodo. La forma más simple de usar R+Slurm es enviar un solo trabajo y solicitar múltiples CPUs para usar, por ejemplo, `parallel::parLapply` o `parallel::mclapply`. Usualmente, los usuarios no necesitan solicitar un número específico de nodos para ser usados ya que Slurm asignará los recursos según sea necesario.

  Un error común de los usuarios de R es especificar el número de nodos y esperar que su script sea paralelizado. Esto no sucederá a menos que el usuario escriba explícitamente un script de computación paralela.

  La bandera relevante para `sbatch` es `--nodes`.

- **Partición** Un grupo de nodos en HPC. Generalmente los nodos grandes pueden tener múltiples particiones, lo que significa que los nodos pueden ser agrupados de varias maneras. Por ejemplo, los nodos pertenecientes a un solo grupo de usuarios pueden estar en una sola partición, y los nodos dedicados a trabajar con datos grandes pueden estar en otra partición. Usualmente, las particiones están asociadas con privilegios de cuenta, así que los usuarios pueden necesitar especificar qué cuenta están usando cuando le dicen a Slurm qué partición planean usar.

  La bandera relevante para `sbatch` es `--partition`.

- **Cuenta** Las cuentas pueden estar asociadas con particiones. Las cuentas pueden tener privilegios para usar una partición o conjunto de nodos. A menudo, los usuarios necesitan especificar la cuenta cuando envían trabajos a una partición particular.

  La bandera relevante para `sbatch` es `--account`.

- **Tarea** Un paso dentro de un trabajo. Un trabajo particular puede tener múltiples tareas. Las tareas pueden abarcar múltiples nodos, así que si el usuario quiere enviar un trabajo multinúcleo, esta opción puede no ser la correcta.

  La bandera relevante para `sbatch` es `--ntasks`

- **CPU** generalmente esto se refiere a núcleo o hilo (que puede ser diferente en sistemas que soportan núcleos multihilo). Los usuarios pueden querer especificar cuántos CPUs quieren usar para una tarea. Y esta es la opción relevante cuando se usan cosas como OpenMP o funciones que permiten crear objetos de clúster en R (p. ej. `makePSOCKcluster`, `makeForkCluster`).

La opción relevante en `sbatch` es `--cpus-per-task`. Más información sobre CPUs en Slurm se puede encontrar aquí. Información sobre cómo Slurm cuenta CPUs/núcleos/hilos se puede encontrar aquí.

- **Array de Trabajos** Slurm soporta arrays de trabajos. Un array de trabajos es en términos simples un trabajo que es repetido múltiples veces por Slurm, esto es, replica un solo trabajo según lo solicitado por el usuario. En el caso de R, cuando se usa esta opción, un solo script R se extiende en múltiples trabajos, así que el usuario puede tomar ventaja de esto y paralelizar trabajos a través de múltiples nodos. Además del hecho de que los trabajos dentro de un Array de Trabajos pueden extenderse a través de múltiples nodos, cada trabajo en ese array tiene un ID único que está disponible para el usuario vía variables de entorno, en particular `SLURM_ARRAY_TASK_ID`.

  Dentro de R, y por tanto el Rscript enviado a Slurm, los usuarios pueden acceder a esta variable de entorno con `Sys.getenv("SLURM_ARRAY_TASK_ID")`. Algunas de las funcionalidades de `slurmR` dependen de Arrays de Trabajos.

  Más información sobre Arrays de Trabajos se puede encontrar aquí. La opción relevante para esto en `sbatch` es `--array`.

Más información sobre Slurm se puede encontrar en su sitio web oficial aquí. Un tutorial sobre cómo usar Slurm con R se puede encontrar aquí.

9. Una introducción breve a Slurm

Para una introducción rápida y sucia a Slurm [4], comenzaremos con un simple "Hola mundo" usando Slurm + R. Para esto, necesitamos pasar por los siguientes pasos:

1. Copiar un script de Slurm a HPC,
2. Iniciar sesión en HPC, y
3. Enviar el trabajo usando `sbatch`.

9.1 Paso 1: Copiar el script de Slurm a HPC

Necesitamos copiar el siguiente script de Slurm a HPC (00-hello-world.slurm):

```
#!/bin/sh
#SBATCH --output=00-hello-world.out
module load R/4.2.2
Rscript -e "paste('Hello from node', Sys.getenv('SLURMD_NODENAME'))"
```

Que tiene cuatro líneas:

1. `#!/bin/sh`: El **shebang** (**¿she qué?**)
2. `#SBATCH --output=00-hello-world.out`: Una opción para ser pasada a `sbatch`, en este caso, el nombre del archivo de salida al cual irán **stdout y stderr**.
3. `module load R/4.2.2`: Usa **Lmod** para cargar el módulo `R`.
4. `Rscript ...`: Una llamada a R para evaluar la expresión `paste(...)`. Esto obtendrá la variable de entorno `SLURMD_NODENAME` (que `sbatch` crea) y la imprimirá en un mensaje.

Para hacer esto, usaremos **Protocolo de copia segura (scp)**, que nos permite copiar datos hacia y desde computadoras. En este caso, deberíamos hacer algo como lo siguiente

```
scp    00-hello-world.slurm    [userid]@notchpeak.chpc.utah.edu:/ruta/a/un/lugar/que/
puedas/acceder
```

En palabras, "Usando el nombre de usuario `[userid]`, conectar a `notchpeak.chpc.utah.edu`, tomar el archivo `00-hello-world.slurm` y copiarlo a `/ruta/a/un/lugar/que/puedas/acceder`. Con el archivo ahora disponible en el clúster, podemos enviar este trabajo usando Slurm.

9.2 Paso 2: Iniciar sesión en HPC

1. Iniciar sesión usando ssh. En el caso de usuarios de Windows, descargar el cliente **Putty**.
2. Para iniciar sesión, necesitarás usar tu ID de organización. Usualmente, si tu email es algo como `miusuarioemail@escuela.edu`, tu ID es `miusuarioemail`. Entonces:

   ```
   ssh miusuarioemail@notchpeak.chpc.utah.edu
   ```

9.3 Paso 3: Enviando el trabajo

En general, hay dos formas de usar los nodos de cómputo: interactivamente (`salloc`) y en modo por lotes (`sbatch`). En este caso, ya que tenemos un script de Slurm, usaremos este último.

Para enviar el trabajo, podemos escribir lo siguiente:

```
sbatch 00-hello-world.slurm
```

¡Y eso es todo! Dicho eso, a menudo se requiere especificar la cuenta y partición que el usuario estará enviando el trabajo. Por ejemplo, si tienes la cuenta `mi-cuenta` y partición `mi-partición` asociadas con tu usuario, puedes incorporar esa información como sigue:

```
sbatch 00-hello-world.slurm --acount=mi-cuenta --partition=mi-partición
```

En el caso de sesiones interactivas, puedes iniciar una usando el comando `salloc`. Por ejemplo, si quisieras ejecutar R con 8 núcleos, usando 16 Gigs de memoria en total, necesitarías hacer lo siguiente:

```
salloc -n1 --cpus-per-task=8 --mem-per-cpu=2G --time=01:00:00
```

Una vez que tu solicitud es enviada, obtendrás acceso a un nodo de cómputo. Dentro de él, puedes cargar los módulos requeridos e iniciar R:

```
module load R/4.2.2
R
```

Las sesiones interactivas no se recomiendan para trabajos largos. En su lugar, usa este recurso si necesitas inspeccionar algún conjunto de datos grande, depurar tu código, etc.

10. SLURM con Simulación de π

⚠ Nota de Traducción

Esta versión del capítulo fue traducida de manera automática utilizando IA. El capítulo aún no ha sido revisado por un humano.

11. Simulando *pi*

El siguiente es un ejemplo que muchas personas (incluyéndome) han usado para ilustrar computación paralela con R. El ejemplo es directo: queremos aproximar pi haciendo algunas simulaciones de Monte Carlo.

Sabemos que el área de un círculo es $A = \pi r^2$, que es equivalente a $\pi = A/r^2$, así que si podemos aproximar el Área de un círculo, entonces podemos aproximar π. ¿Cómo hacemos esto?

Usando experimentos de Monte Carlo, podemos aproximar la probabilidad de que un punto aleatorio x caiga dentro del círculo unitario usando la siguiente fórmula:

$$\hat{p} = \frac{1}{n} \sum_i \mathbf{1}(x \in \text{Círculo}) \tag{11.1}$$

Esta aproximación, $\hat{p}$, multiplicada por el área del cuadrado que contiene al círculo, que tiene un área igual a $(2 \times r)^2$, así, finalmente podemos escribir

$$\hat{\pi} = \hat{p} \times (2 \times r)^2 / r^2 = 4\hat{p} \tag{11.2}$$

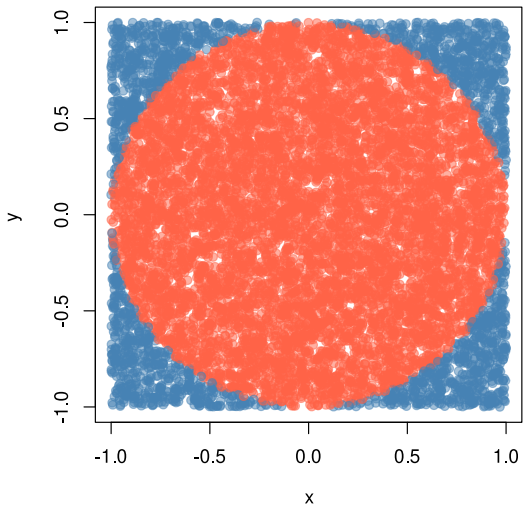


Figure 11.1: 10,000 puntos aleatorios dibujados dentro del círculo unitario.

11.1 Enviando trabajos a Slurm

Trabajaremos principalmente enviando trabajos usando la función `sbatch`. Esta función toma como su argumento principal un archivo bash con el programa a ejecutar. En el caso de R, un archivo bash regular se ve algo así:

```
#!/bin/sh
#SBATCH --job-name=sapply
#SBATCH --time=00:10:00

module load R/4.2.2
Rscript --vanilla 01-sapply.R
```

Este archivo tiene tres componentes:

- **Las banderas de Slurm `#SBATCH`**: Estas entradas parecidas a comentarios pasan opciones de Slurm al trabajo. En este ejemplo, solo especificamos las opciones `job-name` y `time`. Otras opciones comunes incluirían `account` y `partition`.
- **Cargando R `module load R/4.2.2`**: Dependiendo de la configuración de tu sistema, puedes o no necesitar cargar módulos o ejecutar scripts bash antes de poder ejecutar R. En este ejemplo, estamos cargando R versión 4.2.2 usando `LMod` (ver sección anterior).
- **Ejecutando el script R**: Después de especificar opciones de Slurm y cargar lo que necesita ser cargado antes de ejecutar R, estamos usando `RScript` para ejecutar el programa que escribimos.

El envío se hace entonces como sigue:

```
sbatch 01-sapply.slurm
```

Los siguientes ejemplos tienen dos archivos, un script bash y un script R, para ser llamado por Slurm.

11.1.1 Caso 1: Trabajo único, trabajo de un solo núcleo

La forma más básica es enviar un trabajo usando el comando `sbatch`. En este caso, debes tener dos archivos: (1) Un script R y (2) un script bash. p. ej.

Los contenidos del script R (01-sapply.R) son:

```
# Model parameters
nsims <- 1e3
n     <- 1e4

# Function to simulate pi
simpi <- function(i) {

  p <- matrix(runif(n*2, -1, 1), ncol = 2)
  mean(sqrt(rowSums(p^2)) <= 1) * 4

}

# Approximation
set.seed(12322)
ans <- sapply(1:nsims, simpi)

message("Pi: ", mean(ans))

saveRDS(ans, "01-sapply.rds")
```

Los contenidos del archivo bash (01-sapply.slurm) son:

```
#!/bin/sh
#SBATCH --job-name=sapply
#SBATCH --time=00:10:00

module load R/4.2.2
Rscript --vanilla 01-sapply.R
```

11.1.2 Caso 2: Trabajo único, trabajo multinúcleo

Imagina que nos gustaría usar más de un procesador para este trabajo, usando la función `parallel::mclapply` del paquete parallel.[8] Entonces, además de adaptar el código, necesitamos decirle a Slurm que estamos usando más de un núcleo por tarea, como en el siguiente ejemplo:

Script R (02-mclapply.R):

[8]Esta función es una especie de envoltorio de `makeForkcluster`. Forking proporciona una forma de duplicar un proceso en el SO sin replicar la memoria, lo cual es tanto más rápido como eficiente.

```
# Model parameters
nsims  <- 1e3
n      <- 1e4
ncores <- 4L

# Function to simulate pi
simpi <- function(i) {

  p <- matrix(runif(n*2, -1, 1), ncol = 2)
  mean(sqrt(rowSums(p^2)) <= 1) * 4

}

# Approximation
set.seed(12322)
ans <- parallel::mclapply(1:nsims, simpi, mc.cores = ncores)
ans <- unlist(ans)

message("Pi: ", mean(ans))

saveRDS(ans, "02-mclpply.rds")
```

Archivo bash (02-mclapply.slurm):

```
#!/bin/sh
#SBATCH --job-name=mclapply
#SBATCH --time=00:10:00
#SBATCH --cpus-per-task=4

module load R/4.2.2
Rscript --vanilla 02-mclapply.R
```

11.2 Trabajos con el paquete slurmR

El paquete R `slurmR` [5], [6] es un envoltorio ligero de Slurm. Las funciones principales del paquete son la familia `*apply`–principalmente a través de arrays de trabajos Slurm–y `makeSlurmCluster()`–que es un envoltorio de `makePSOCKcluster`.

Esta sección ilustrará cómo enviar trabajos usando la función `makeSlurmCluster()` y `Slurm_sapply`. Además, el último ejemplo demuestra cómo podemos omitir escribir scripts Slurm completamente usando la función `sourceSlurm()` incluida en el paquete.

11.2.1 Caso 3: Trabajo único, trabajo multinodo

En este caso, no hay una forma simple de enviar un trabajo multinodal a Slurm; a menos que uses el paquete **slurmR**.[9] En este ejemplo, combinaremos `slurmR` con la función `parSapply` del paquete parallel para enviar un trabajo multinodal usando la función `makeSlurmCluster()`. Con ella, `slurmR` enviará un trabajo solicitando `njobs` tareas (procesadores) que podrían abarcar múltiples nodos,[10] y crear un clúster Socket de él (como usar `makePSOCKcluster`.) Una cosa a tener en mente es que los clústeres Socket están limitados en el número de conexiones que una sola sesión R puede abarcar. Puedes leer más sobre eso aquí y aquí.

Script R (03-parsapply-slurmr.R):

```
# Model parameters
nsims  <- 1e3
n      <- 1e4
ncores <- 4L
```

[9] Ver instrucciones de instalación **aquí**

[10] Aunque posible, la mayoría de trabajos multinodo serán asignados si no hay suficientes hilos dentro de un solo nodo. Recuerda que Slurm no ejecuta los trabajos sino que reserva recursos computacionales para que los ejecutes.

```
# Function to simulate pi
simpi <- function(i) {

  p <- matrix(runif(n*2, -1, 1), ncol = 2)
  mean(sqrt(rowSums(p^2)) <= 1) * 4

}

# Setting up slurmR
library(slurmR) # This also loads the parallel package

# Making the cluster, and exporting the variables
cl <- makeSlurmCluster(ncores)

# Approximation
clusterExport(cl, c("n", "simpi"))
ans <- parSapply(cl, 1:nsims, simpi)

# Closing connection
stopCluster(cl)

message("Pi: ", mean(ans))

saveRDS(ans, "03-parsapply-slurmr.rds")
```

Archivo bash (03-parsapply-slurmr.slurm):

```
#!/bin/sh
#SBATCH --job-name=parsapply
#SBATCH --time=00:10:00

module load R/4.2.2
Rscript --vanilla 03-parsapply-slurmr.R
```

11.2.2 Caso 4: Múltiples trabajos, un solo/múltiples núcleos

Otra forma de enviar trabajos es usando **arrays de trabajos**. Un array de trabajos es un trabajo repetido `njobs` veces con la misma configuración. La principal diferencia entre réplicas es lo que haces con la variable de entorno `SLURM_ARRAY_TASK_ID`. Esta variable se define dentro de cada réplica y puede usarse para hacer el "subtrabajo" dependiendo de eso.

Aquí hay un ejemplo rápido usando R

```
ID <- Sys.getenv("SLURM_ARRAY_TASK_ID")
if (ID == 1) {
  ...[haz esto]...
} else if (ID == 2) {
  ...[haz eso]...
}
```

El paquete R `slurmR` hace fácil enviar arrays de trabajos. Nuevamente, con la simulación de pi, podemos hacerlo de la siguiente manera:

Script R (04-slurm_sapply.R):

```
# Model parameters
nsims  <- 1e3
n      <- 1e4
# ncores <- 4L
njobs  <- 4L

# Function to simulate pi
```

```
simpi <- function(i, n.) {

  p <- matrix(runif(n.*2, -1, 1), ncol = 2)
  mean(sqrt(rowSums(p^2)) <= 1) * 4

}

# Setting up slurmR
library(slurmR) # This also loads the parallel package

# Approximation
ans <- Slurm_sapply(
  1:nsims, simpi,
  n.       = n,
  njobs    = njobs,
  plan     = "collect",
  tmp_path = "/scratch/vegayon" # This is where all temp files will be exported
  )

message("Pi: ", mean(ans))

saveRDS(ans, "04-slurm_sapply.rds")
```

Archivo bash (04-slurm_sapply.slurm):

```
#!/bin/sh
#SBATCH --job-name=slurm_sapply
#SBATCH --time=00:10:00

module load R/4.2.2
Rscript --vanilla 04-slurm_sapply.R
```

Uno de los principales beneficios de usar este enfoque en lugar de la función `makeSlurmCluster` (y así, trabajar con un clúster SOCK) son:

- El número de trabajos no está limitado aquí (solo por el administrador, pero no por R).
- Si un trabajo falla, entonces podemos volver a ejecutarlo usando `sbatch` una vez más (ver ejemplo aquí).
- Puedes verificar los logs individuales de cada proceso usando la función `Slurm_lob()`.
- Puedes enviar el trabajo y salir de la sesión R sin esperar a que finalice. Siempre puedes leer de vuelta el trabajo usando la función `read_slurm_job([ruta-al-temp])`

11.2.3 Caso 5: Omitiendo el archivo .slurm

El paquete `slurmR` tiene una función llamada `sourceSlurm` que puede usarse para evitar crear el archivo `.slurm`. El usuario puede agregar las opciones SBATCH al principio del script R (incluyendo la línea `#!/bin/sh`) y enviar el trabajo desde R como sigue:

Script R (05-sapply.R):

```
#!/bin/sh
#SBATCH --job-name=sapply-sourceSlurm
#SBATCH --time=00:10:00

# Model parameters
nsims <- 1e3
n     <- 1e4

# Function to simulate pi
simpi <- function(i) {

  p <- matrix(runif(n*2, -1, 1), ncol = 2)
```

```
  mean(sqrt(rowSums(p^2)) <= 1) * 4

}

# Approximation
set.seed(12322)
ans <- sapply(1:nsims, simpi)

message("Pi: ", mean(ans))

saveRDS(ans, "05-sapply.rds")
```

Desde la consola R (está bien si estás en el nodo cabeza)

```
slurmR::sourceSlurm("05-sapply.R")
```

¡Y voilá! Se generará un archivo bash temporal para enviar el script R a la cola. El siguiente video muestra una posible salida en el CHPC de la Universidad de Utah con slurmR versión 0.5-3:

https://youtu.be/OasEla5EszI

IV Usando C++

12. Rcpp

⚠ Nota de Traducción

Esta versión del capítulo fue traducida de manera automática utilizando IA. El capítulo aún no ha sido revisado por un humano.

Cuando la computación paralela no es suficiente, puedes acelerar tu código R usando un lenguaje de programación de más bajo nivel[11] como C++, C, o Fortran. Con R mismo escrito en C, proporciona puntos de acceso (APIs) para conectar funciones de C++/C/Fortran a R. Aunque no es imposible, usar lenguajes de bajo nivel para mejorar R puede ser engorroso; Rcpp [7], [8], [9] puede hacer las cosas **muy** fáciles. Este capítulo te muestra cómo usar Rcpp–la forma más popular de conectar C++ con R–para acelerar tu código R.

12.1 Antes de empezar

1. Necesitas tener Rcpp instalado en tu sistema:

```
install.packages("Rcpp")
```

2. Necesitas tener un compilador
   - Windows: Puedes descargar Rtools desde aquí.
   - MacOS: Es un poco complicado... Aquí hay algunas opciones:
     - Manual de CRAN para obtener los compiladores clang, clang++, y gfortran aquí.
     - Una excelente guía del profesor coatless aquí

¡Y eso es todo!

12.2 R es genial, pero...

- El problema:
  - Como vimos, R es muy rápido... una vez vectorizado
  - ¿Qué hacer si tu modelo no puede ser vectorizado?
- La solución: **¡Usa C/C++/Fortran! ¡Funciona con R!**
- El problema de la solución: **¿Qué usuario de R conoce alguno de esos?**
- R ha tenido una API (interfaz de programación de aplicaciones) para integrar código C/C++ con R durante mucho tiempo.
- Desafortunadamente, no es muy directo

12.3 Entra Rcpp

- Uno de los **paquetes de R más importantes en CRAN**.

[11] En general, un lenguaje de programación de bajo nivel es "*un lenguaje de programación que proporciona poca o ninguna abstracción del conjunto de arquitectura de una computadora [...]*" (wiki), sin embargo, aquí usamos ese término para referirnos a lenguajes de programación que están más cerca del código máquina de lo que está R.

- Al 22 de enero de 2023, aproximadamente 50% de los paquetes de CRAN dependen de él (directa o indirectamente).
- De la descripción del paquete:

> El paquete 'Rcpp' proporciona funciones de R así como clases de C++ que ofrecen una integración perfecta de R y C++

12.4 ¿Por qué molestarse?

- Para extraer diez números de una distribución normal con sd = 100.0 usando la API de C de R:

```
SEXP stats = PROTECT(R_FindNamespace(mkString("stats")));
SEXP rnorm = PROTECT(findVarInFrame(stats, install("rnorm")));
SEXP call = PROTECT(
  LCONS( rnorm, CONS(ScalarInteger(10), CONS(ScalarReal(100.0),
  R_NilValue))));
SET_TAG(CDDR(call),install("sd"));
SEXP res = PROTECT(eval(call, R_GlobalEnv));
UNPROTECT(4);
return res;
```

- Usando Rcpp:

```
Environment stats("package:stats");
Function rnorm = stats["rnorm"];
return rnorm(10, Named("sd", 100.0));
```

12.5 Ejemplo 1: Iterando sobre un vector

```
#include<Rcpp.h>
using namespace Rcpp;
// [[Rcpp::export]]
NumericVector add1(NumericVector x) {
  NumericVector ans(x.size());
  for (int i = 0; i < x.size(); ++i)
    ans[i] = x[i] + 1;
  return ans;
}
```

```
add1(1:10)
```

```
 [1]  2  3  4  5  6  7  8  9 10 11
```

Hazlo más dulce agregando algo de "azúcar" (del tipo Rcpp)

```
#include<Rcpp.h>
using namespace Rcpp;
// [[Rcpp::export]]
NumericVector add1Cpp(NumericVector x) {
  return x + 1;
}
```

```
add1Cpp(1:10)
```

```
 [1]  2  3  4  5  6  7  8  9 10 11
```

12.6 ¿Qué tan rápido?

Comparado con esto:

```
add1R <- function(x) {
  for (i in 1:length(x))
    x[i] <- x[i] + 1
  x
}
microbenchmark::microbenchmark(add1R(1:1000), add1Cpp(1:1000))
```

```
Unit: microseconds
            expr    min      lq     mean  median      uq      max neval
   add1R(1:1000) 60.302 60.6680 84.72882 61.1540 64.0795 2279.733   100
 add1Cpp(1:1000)  2.795  3.1005 12.25253  3.4215  9.0920  684.557   100
```

12.7 Principales diferencias entre R y C++

1. Uno es compilado, y el otro interpretado
2. Indexando objetos: En C++ los índices van de 0 a `(n - 1)`, mientras que en R es de 1 a `n`.
3. Todas las expresiones terminan con `;` (opcional en R).
4. En C++ los objetos necesitan ser declarados, en R no (dinámico).

12.8 Fundamentos de C++/Rcpp: Tipos

Además de tipos de datos tipo C (`double`, `int`, `char`, y `bool`), podemos usar los siguientes tipos de objetos con Rcpp:

- Matrices: `NumericMatrix`, `IntegerMatrix`, `LogicalMatrix`, `CharacterMatrix`
- Vectores: `NumericVector`, `IntegerVector`, `LogicalVector`, `CharacterVector`
- ¡Y más!: `DataFrame`, `List`, `Function`, `Environment`

12.9 Partes de "un programa Rcpp"

```
#include<Rcpp.h>
using namespace Rcpp
// [[Rcpp::export]]
NumericVector add1(NumericVector x) {
  NumericVector ans(x.size());
  for (int i = 0; i < x.size(); ++i)
    ans[i] = x[i] + 1;
  return ans;
}
```

Línea por línea, vemos lo siguiente:

1. El #include<Rcpp.h> es similar a `library(...)` en R, trae todo lo que necesitamos para escribir código C++ para Rcpp.
2. using namespace Rcpp es algo similar a `detach(...)`. Esto simplifica la sintaxis. Si no incluimos esto, todas las llamadas a miembros de Rcpp necesitan ser explícitas, **ej.**, en lugar de escribir `NumericVector`, necesitaríamos escribir `Rcpp::NumericVector`
3. El `//` comienza un comentario en C++, en este caso, el // [[Rcpp::export]] comentario es una bandera que Rcpp usa para "exportar" esta función C++ a R.
4. Es la primera parte de la definición de función. Estamos creando una función que devuelve un NumericVector, se llama add1, tiene un solo elemento de entrada llamado x que también es un NumericVector.

5. Aquí, estamos declarando un objeto llamado ans, que es un NumericVector con un tamaño inicial igual al tamaño de x. Nota que .size() se llama una "función miembro" del objeto `x`, que es de clase `NumericVector`.
6. Estamos declarando un for-loop (tres partes):
   a. int i = 0 Declaramos la variable `i`, un entero, y la inicializamos en 0.
   b. i < x.size() Este bucle terminará cuando el valor de `i` esté en o por encima de la longitud de `x`.
   c. ++i En cada iteración, `i` se incrementará en una unidad.
7. ans[i] = x[i] + 1 establece el i-ésimo elemento de `ans` igual al i-ésimo elemento de `x` más 1.
8. return ans sale de la función devolviendo el vector `ans`.

Ahora, ¿dónde ejecutar/correr esto?

- Puedes usar la función `sourceCpp` del paquete `Rcpp` para ejecutar scripts .cpp (esto es lo que hago la mayoría del tiempo).
- También está `cppFunction`, que permite compilar una sola función.
- Escribir un paquete de R que funcione con Rcpp.

Por ahora, usemos la primera opción.

12.10 Ejemplo ejecutando archivo .cpp

Imagina que tenemos el siguiente archivo llamado `norm.cpp`

```
#include <Rcpp.h>
using namespace Rcpp;
// [[Rcpp::export]]
double normRcpp(NumericVector x) {

  return sqrt(sum(pow(x, 2.0)));

}
```

Podemos compilar y obtener esta función usando esta línea `Rcpp::sourceCpp("norm.cpp")`. Una vez compilada, una función llamada `normRcpp` estará disponible en la sesión actual de R.

12.11 Tu turno

12.11.1 Problema 1: Sumar vectores

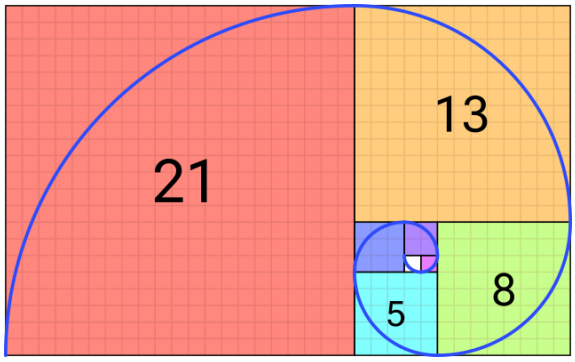
1. Usando lo que acabas de aprender sobre Rcpp, escribe una función para sumar dos vectores de la misma longitud. Usa la siguiente plantilla

```
#include <Rcpp.h>
using namespace Rcpp;
// [[Rcpp::export]]
NumericVector add_vectors([declarar vector 1], [declarar vector 2]) {

  ... magia ...

  return [algo];
}
```

2. Ahora, tenemos que verificar las longitudes. Usa la función `stop` para asegurarte de que las longitudes coincidan. Agrega las siguientes líneas en tu código

```
if ([alguna condición])
  stop("un mensaje de error arbitrario :)");
```

12.11.2 Problema 2: Serie de Fibonacci



Cada elemento de la secuencia está determinado por lo siguiente:

$$F(n) = \begin{cases} n, & \text{si } \ n \leq 1 \\ F(n-1) + F(n-2), & \text{de otro modo} \end{cases} \tag{12.1}$$

Usando recursiones, podemos implementar este algoritmo en R de la siguiente manera:

```
fibR <- function(n) {
  if (n <= 1)
    return(n)
  fibR(n - 1) + fibR(n - 2)
}
# ¿Está funcionando?
c(
  fibR(0), fibR(1), fibR(2),
  fibR(3), fibR(4), fibR(5),
  fibR(6)
)
```

```
[1] 0 1 1 2 3 5 8
```

Ahora, ¡traduzcamos este código a Rcpp y veamos cuánto aumento de velocidad obtenemos!

12.11.3 Problema 2: Serie de Fibonacci (solución)

```
#include <Rcpp.h>
// [[Rcpp::export]]
int fibCpp(int n) {
  if (n <= 1)
    return n;

  return fibCpp(n - 1) + fibCpp(n - 2);

}
```

```
microbenchmark::microbenchmark(fibR(20), fibCpp(20))
```

```
Unit: microseconds
       expr      min       lq       mean    median        uq      max neval
   fibR(20) 6214.631 6256.735 6461.58646 6274.0175 6404.3905 8247.644   100
 fibCpp(20)   15.780   16.025   26.22053   16.9365   17.5675  895.170   100
```

12.12 RcppArmadillo y OpenMP

- Más amigable que **RcppParallel**... al menos para usuarios 'Uso-Rcpp-pero-realmente-no-sé-mucho-sobre-C++' (¡como yo!).
- Debe ejecutar solo llamadas 'Thread-safe', así que llamar R dentro de bloques paralelos puede causar problemas (casi todo el tiempo).
- Usa objetos `arma`, ej. `arma::mat`, `arma::vec`, etc. O, si estás acostumbrado a ellos objetos `std::vector` ya que estos son thread-safe.
- La Generación de Números Pseudo Aleatorios no es muy directa... Pero C++11 tiene un buen conjunto de funciones que pueden usarse junto con OpenMP
- Necesitas pensar sobre cómo funcionan los procesadores, memoria caché, etc. De otro modo, podrías meterte en problemas... si tu código es más lento cuando se ejecuta en paralelo, entonces probablemente estés enfrentando false sharing
- Si R se cuelga... intenta ejecutar R con un debugger (ver Sección 4.3 en Writing R extensions):

```
~$ R --debugger=valgrind
```

12.12.1 Flujo de trabajo de RcppArmadillo y OpenMP

1. Dile a Rcpp que necesitas incluir eso en el compilador:

```
#include <omp.h>
// [[Rcpp::plugins(openmp)]]
```

2. Dentro de tu función, establece el número de núcleos, ej.

```
// Estableciendo los núcleos
omp_set_num_threads(cores);
```

3. Dile al compilador que ejecutarás un bloque en paralelo con OpenMP

```
#pragma omp [directivas] [opciones]
{
  ...tu código paralelo elegante...
}
```

   Necesitarás especificar cómo OMP debe manejar los datos:

   - `shared`: Por defecto, todos los hilos acceden a la misma copia.
   - `private`: Cada hilo tiene su propia copia, no inicializada.
   - `firstprivate` Cada hilo tiene su propia copia, inicializada.
   - `lastprivate` Cada hilo tiene su propia copia. El último valor usado se devuelve.

   Establecer `default(none)` es una buena práctica.

4. ¡Compila!

12.12.2 Ej 5: RcppArmadillo + OpenMP

Nuestra propia versión de la función `dist`... ¡pero en paralelo!

```
#include <omp.h>
#include <RcppArmadillo.h>
// [[Rcpp::depends(RcppArmadillo)]]
// [[Rcpp::plugins(openmp)]]
using namespace Rcpp;
// [[Rcpp::export]]
arma::mat dist_par(const arma::mat & X, int cores = 1) {
```

```
  // Algunas constantes
  int N = (int) X.n_rows;
  int K = (int) X.n_cols;

  // Salida
  arma::mat D(N,N);
  D.zeros(); // Rellenando con ceros

  // Estableciendo los núcleos
  omp_set_num_threads(cores);

#pragma omp parallel for shared(D, N, K, X) default(none)
  for (int i=0; i<N; ++i)
    for (int j=0; j<i; ++j) {
      for (int k=0; k<K; k++)
        D.at(i,j) += pow(X.at(i,k) - X.at(j,k), 2.0);

      // Calculando raíz cuadrada
      D.at(i,j) = sqrt(D.at(i,j));
      D.at(j,i) = D.at(i,j);
    }


  // Mi bonita matriz de distancia
  return D;
}
```

```
# Simulando datos
set.seed(1231)
K <- 5000
n <- 500
x <- matrix(rnorm(n*K), ncol=K)
# ¿Estamos obteniendo lo mismo?
table(as.matrix(dist(x)) - dist_par(x, 4)) # Solo ceros
```

```
     0
250000
```

```
# ¡Benchmarking!
microbenchmark::microbenchmark(
  dist(x),                # stats::dist
  dist_par(x, cores = 1), # 1 núcleo
  dist_par(x, cores = 2), # 2 núcleos
  dist_par(x, cores = 4), # 4 núcleos
  times = 1,
  unit = "ms"
)
```

```
Unit: milliseconds
                  expr       min        lq      mean    median        uq
               dist(x) 1172.8309 1172.8309 1172.8309 1172.8309 1172.8309
 dist_par(x, cores = 1)  855.6636  855.6636  855.6636  855.6636  855.6636
 dist_par(x, cores = 2)  678.0219  678.0219  678.0219  678.0219  678.0219
 dist_par(x, cores = 4)  623.4489  623.4489  623.4489  623.4489  623.4489
       max neval
 1172.8309     1
  855.6636     1
```

```
  678.0219     1
  623.4489     1
```

12.12.3 Ej 6: El futuro

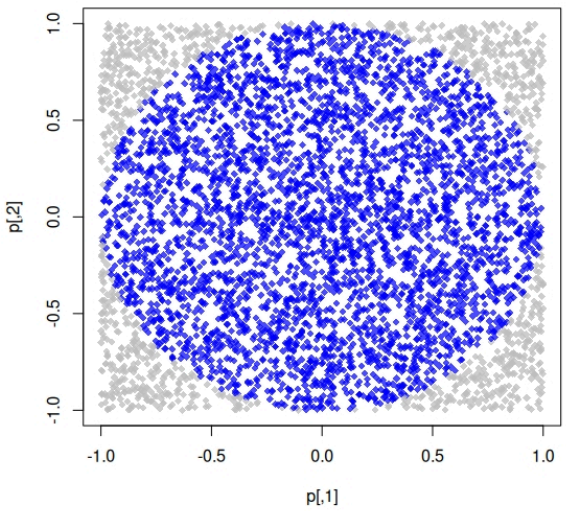
- **future** es un paquete de R que fue diseñado "para proporcionar una forma muy simple y uniforme de evaluar expresiones de R de forma asíncrona usando varios recursos disponibles para el usuario."
- Los objetos de clase `future` están resueltos o no resueltos.
- Si se consultan, los valores **Resueltos** se devuelven inmediatamente, y los valores **No resueltos** bloquearán el proceso (es decir, esperarán) hasta que se resuelvan.
- Los futures pueden ser paralelos/seriales, en una sola computadora (local o remota), o un cluster de ellas.

Veamos un breve ejemplo

```
library(future)
plan(multicore)
# Estamos creando una variable global
a <- 2
# Crear los futures solo tiene el tiempo de sobrecarga (configuración)
system.time({
  x1 %<-% {Sys.sleep(3);a^2}
  x2 %<-% {Sys.sleep(3);a^3}
})
##    user  system elapsed
##   0.029   0.015   0.043
# Esperemos solo 5 segundos para asegurarnos de que todos los núcleos hayan regresado
Sys.sleep(3)
system.time({
  print(x1)
  print(x2)
})
## [1] 4
## [1] 8
##    user  system elapsed
##   0.005   0.000   0.004
```

12.12.4 Pista bonus 1: Simulando π

- Sabemos que $\pi = \frac{A}{r^2}$. Lo aproximamos agregando aleatoriamente puntos x a un cuadrado de tamaño 2 centrado en el origen.
- Entonces, aproximamos π como $\Pr\{\| x \| \leq 1\} \times 2^2$



El código R para hacer esto

```
pisim <- function(i, nsim) {  # Nota que no usamos la -i-
  # Puntos aleatorios
  ans  <- matrix(runif(nsim*2), ncol=2)

  # Distancia al origen
  ans  <- sqrt(rowSums(ans^2))

  # Pi estimado
  (sum(ans <= 1)*4)/nsim
}
```

```
library(parallel)
# Configuración
cl <- makePSOCKcluster(4L)
clusterSetRNGStream(cl, 123)
# Número de simulaciones que queremos que cada una ejecute
nsim <- 1e5
# Necesitamos hacer que -nsim- y -pisim- estén disponibles para el
# cluster
clusterExport(cl, c("nsim", "pisim"))
# Benchmarking: parSapply y sapply ejecutarán esta simulación
# cien veces cada una, así que al final tenemos 1e5*100 puntos
# para aproximar pi
microbenchmark::microbenchmark(
  parallel = parSapply(cl, 1:100, pisim, nsim=nsim),
  serial   = sapply(1:100, pisim, nsim=nsim),
  times    = 1
)
```

```
Unit: milliseconds
     expr      min       lq     mean   median       uq      max neval
 parallel 290.3510 290.3510 290.3510 290.3510 290.3510 290.3510     1
   serial 377.0268 377.0268 377.0268 377.0268 377.0268 377.0268     1
```

```
ans_par <- parSapply(cl, 1:100, pisim, nsim=nsim)
ans_ser <- sapply(1:100, pisim, nsim=nsim)
stopCluster(cl)
```

```
     par      ser        R
3.141762 3.141266 3.141593
```

12.13 Ver también

- Package parallel
- Using the iterators package
- Using the foreach package
- 32 OpenMP traps for C++ developers
- The OpenMP API specification for parallel programming
- 'openmp' tag in Rcpp gallery
- OpenMP tutorials and articles

Para más, revisa el CRAN Task View on HPC

13. Depuración de R con código C++/C

⚠ Nota de Traducción

Esta versión del capítulo fue traducida de manera automática utilizando IA. El capítulo aún no ha sido revisado por un humano.

14. Depuración de R con código C++/C

Aunque depurar código R es fácil, lo mismo no aplica para código compilado en R[12]. Este capítulo muestra algunas formas de depurar tu código R + C++. Necesitarás el GNU Debugger (GDB) y Valgrind.

Antes de comenzar, recuerda que no vamos a lidiar con el buen enfoque de toda la vida `Rprint("Tu código está funcionando hasta aquí")`. Imprimir mensajes mientras tu programa se ejecuta puede ser muy informativo, pero usar Valgrind y GDB es, en mi humilde opinión, más rápido ya que, la mayoría del tiempo, esos te gritarán, indicando la ubicación de tu problema.

> **Note**
>
> El manual Writing R Extensions [1] tiene una cantidad considerable de información sobre depurar código compilado en R aquí. Dirk Eddelbuettel (autor principal de Rcpp) tiene un excelente post en Stackoverflow y recomienda un tutorial alojado en BioConductor [10].

14.1 Depuración con Valgrind

Como punto de partida, usaremos Valgrind. Valgrind proporciona un marco de trabajo maduro para depuración y perfilado de memoria. Debemos lanzar el programa a través de la línea de comandos para usar un depurador dentro de R. Para lanzar R con Valgrind, usamos lo siguiente:

```
$ R --debugger=valgrind
```

Lo cual resultará en algo como lo siguiente:

```
==31245== Memcheck, a memory error detector
==31245== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==31245== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==31245== Command: /usr/lib/R/bin/exec/R
==31245==

R version 4.2.3 (2023-03-15) -- "Shortstop Beagle"
Copyright (C) 2023 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

[12] Depurar **solo** código C++/C es fácil, sin embargo. Si ya trabajas con código compilado, debes estar consciente de VS Code y las muchas otras herramientas por ahí para depurar código C++/C.

Una vez que R se ejecuta con Valgrind, el depurador capturará cualquier fuga de memoria generada por tu código C++/C. Lo siguiente es un programa defectuoso de Rcpp que crea un puntero usando `new` y "olvida" eliminarlo.

```
#include <Rcpp.h>

using namespace Rcpp;

// [[Rcpp::export]]
NumericVector faulty_program(int n) {

    // Aquí está la línea defectuosa
    NumericVector * x_ptr = new NumericVector(n);

    return *x_ptr;

}

/***R
# Calling the faulty program
faulty_program(10)
*/
```

Podemos usar la bandera `-e` en el comando `R` para compilar el script `Rcpp` usando `sourceCpp`:

```
R --debugger=valgrind -e 'Rcpp::sourceCpp("rcpp-debugging-faulty.cpp")'
```

```
==1806== Memcheck, a memory error detector
==1806== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==1806== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==1806==  Command:  /usr/local/lib/R/bin/exec/R  -e  Rcpp::sourceCpp("rcpp-debugging-
faulty.cpp")
==1806==

R version 4.5.3 (2026-03-11) -- "Reassured Reassurer"
Copyright (C) 2026 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> Rcpp::sourceCpp("rcpp-debugging-faulty.cpp")

> faulty_program(10)
 [1] 0 0 0 0 0 0 0 0 0 0
>
==1806==
==1806== HEAP SUMMARY:
==1806==     in use at exit: 60,883,195 bytes in 11,665 blocks
==1806==   total heap usage: 31,690 allocs, 20,025 frees, 93,845,282 bytes allocated
```

```
==1806==
==1806== LEAK SUMMARY:
==1806==    definitely lost: 32 bytes in 1 blocks
==1806==    indirectly lost: 0 bytes in 0 blocks
==1806==      possibly lost: 0 bytes in 0 blocks
==1806==    still reachable: 60,883,163 bytes in 11,664 blocks
==1806==                       of which reachable via heuristic:
==1806==                         newarray           : 4,264 bytes in 1 blocks
==1806==         suppressed: 0 bytes in 0 blocks
==1806== Rerun with --leak-check=full to see details of leaked memory
==1806==
==1806== For lists of detected and suppressed errors, rerun with: -s
==1806== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Al final de la salida, en la sección `LEAK SUMMARY`, vemos `definitely lost: 24 bytes in 1 block`, *es decir*, una fuga de memoria. Si cambiamos el programa eliminando el puntero antes de retornar, la fuga se resolverá:
Nuevo programa:

```
    NumericVector res = *x_ptr;
    delete x_ptr;

    return res;
```

Programa anterior

```
    return *x_ptr;
```

Re-ejecutando R con Valgrind retorna lo siguiente (solo las últimas pocas líneas):

```
R --debugger=valgrind -e 'Rcpp::sourceCpp("rcpp-debugging-faulty-fixed.cpp")'
```

```
==1863== HEAP SUMMARY:
==1863==     in use at exit: 60,883,841 bytes in 11,664 blocks
==1863==   total heap usage: 31,727 allocs, 20,063 frees, 93,865,727 bytes allocated
==1863==
==1863== LEAK SUMMARY:
==1863==    definitely lost: 0 bytes in 0 blocks
==1863==    indirectly lost: 0 bytes in 0 blocks
==1863==      possibly lost: 0 bytes in 0 blocks
==1863==    still reachable: 60,883,841 bytes in 11,664 blocks
==1863==                       of which reachable via heuristic:
==1863==                         newarray           : 4,264 bytes in 1 blocks
==1863==         suppressed: 0 bytes in 0 blocks
==1863== Rerun with --leak-check=full to see details of leaked memory
==1863==
==1863== For lists of detected and suppressed errors, rerun with: -s
==1863== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

No más fugas de memoria.

14.2 Usando GDB

A veces, necesitamos ir más allá e inspeccionar qué está pasando **dentro** del programa. GBD es excelente para eso. Con GBD, podemos establecer puntos de interrupción que nos permiten revisar el programa mientras se ejecuta.

El siguiente código Rcpp genera un error de tipo `memory not mapped`:

```
#include <Rcpp.h>

using namespace Rcpp;

// [[Rcpp::export]]
NumericVector faulty_program(int n) {

    // Aquí está la línea defectuosa
    NumericVector * x_ptr;

    return *x_ptr;

}

/***R
# Calling the faulty program
faulty_program(10)
*/
```

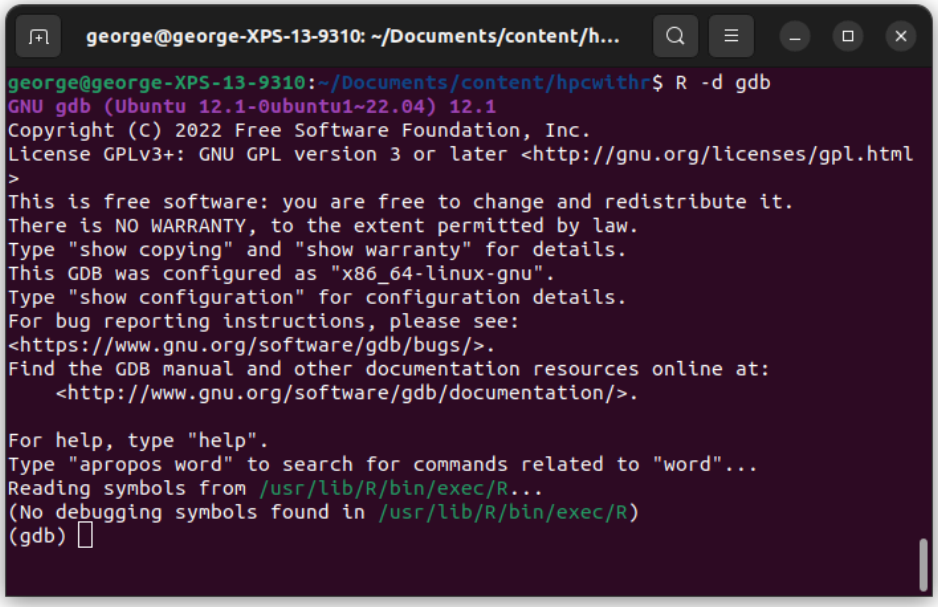
En él, tratamos de acceder a una ubicación en la memoria que no ha sido asignada aún, es decir, un `NumericVector` declarado como un puntero pero nunca asignado. Usar `R --debugger=valgrind` genera el siguiente código:

```
==1920== Memcheck, a memory error detector
==1920== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==1920== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==1920== Command: /usr/local/lib/R/bin/exec/R -e Rcpp::sourceCpp("rcpp-debugging-not-
mapped.cpp")
==1920==

R version 4.5.3 (2026-03-11) -- "Reassured Reassurer"
Copyright (C) 2026 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> Rcpp::sourceCpp("rcpp-debugging-not-mapped.cpp")

> faulty_program(10)
==1920== Invalid read of size 8
==1920==    at 0x2F34F5D6: get__ (PreserveStorage.h:52)
==1920==         by  0x2F34F5D6:  copy__<Rcpp::Vector<14,  Rcpp::PreserveStorage>  >
(PreserveStorage.h:66)
==1920==    by 0x2F34F5D6: Vector (Vector.h:64)
==1920==    by 0x2F34F5D6: faulty_program(int) (rcpp-debugging-not-mapped.cpp:11)
==1920==          by  0x2F34F758:  sourceCpp_1_faulty_program  (rcpp-debugging-not-
mapped.cpp:33)
==1920==    by 0x495DB7D: R_doDotCall (dotcode.c:754)
==1920==    by 0x495E0F6: do_dotcall (dotcode.c:1437)
```

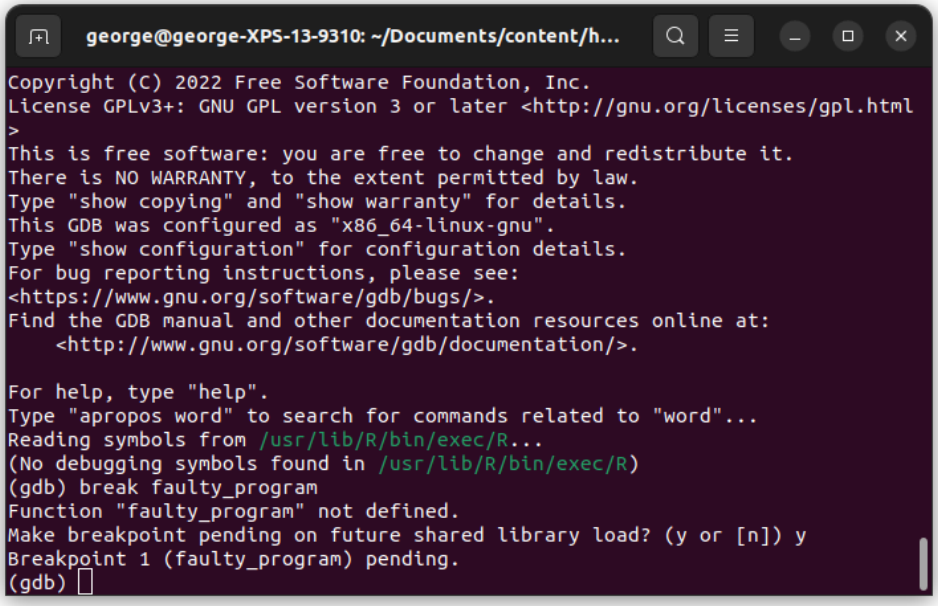
```
==1920==    by 0x49ACA5C: Rf_eval (eval.c:1260)
==1920==    by 0x49AE67D: R_execClosure (eval.c:2393)
==1920==    by 0x49AF3D6: applyClosure_core (eval.c:2306)
==1920==    by 0x49AC575: Rf_applyClosure (eval.c:2328)
==1920==    by 0x49AC575: Rf_eval (eval.c:1280)
==1920==    by 0x49B30E3: do_eval (eval.c:3959)
==1920==    by 0x499F3E4: bcEval_loop (eval.c:8118)
==1920==    by 0x49AC069: bcEval (eval.c:7501)
==1920==    by 0x49AC069: bcEval (eval.c:7486)
==1920==    by 0x49AC43A: Rf_eval (eval.c:1167)
==1920==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
==1920==

 *** caught segfault ***
address (nil), cause 'memory not mapped'

Traceback:
 1: .Call(<pointer: 0x2f34f6e0>, n)
 2: faulty_program(10)
 3: eval(ei, envir)
 4: eval(ei, envir)
 5: withVisible(eval(ei, envir))
 6: source(file = srcConn, local = env, echo = echo)
 7: Rcpp::sourceCpp("rcpp-debugging-not-mapped.cpp")
An irrecoverable exception occurred. R is aborting now ...
==1920==
==1920== Process terminating with default action of signal 11 (SIGSEGV): dumping core
==1920==    at 0x4D6BB2C: __pthread_kill_implementation (pthread_kill.c:44)
==1920==    by 0x4D6BB2C: __pthread_kill_internal (pthread_kill.c:78)
==1920==    by 0x4D6BB2C: pthread_kill@@GLIBC_2.34 (pthread_kill.c:89)
==1920==    by 0x4D1227D: raise (raise.c:26)
==1920==    by 0x4D1232F: ??? (in /usr/lib/x86_64-linux-gnu/libc.so.6)
==1920==         by  0x2F34F5D5:  copy__<Rcpp::Vector<14,  Rcpp::PreserveStorage>  >
(PreserveStorage.h:65)
==1920==    by 0x2F34F5D5: Vector (Vector.h:64)
==1920==    by 0x2F34F5D5: faulty_program(int) (rcpp-debugging-not-mapped.cpp:11)
==1920==
==1920== HEAP SUMMARY:
==1920==     in use at exit: 60,975,521 bytes in 11,879 blocks
==1920==   total heap usage: 31,659 allocs, 19,780 frees, 93,822,386 bytes allocated
==1920==
==1920== LEAK SUMMARY:
==1920==    definitely lost: 0 bytes in 0 blocks
==1920==    indirectly lost: 0 bytes in 0 blocks
==1920==      possibly lost: 5,987 bytes in 19 blocks
==1920==    still reachable: 60,969,534 bytes in 11,860 blocks
==1920==                       of which reachable via heuristic:
==1920==                         newarray           : 4,264 bytes in 1 blocks
==1920==         suppressed: 0 bytes in 0 blocks
==1920== Rerun with --leak-check=full to see details of leaked memory
==1920==
==1920== For lists of detected and suppressed errors, rerun with: -s
==1920== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
Segmentation fault (core dumped)
```

Para inspeccionar un error con GDB, tenemos que seguir estos pasos:

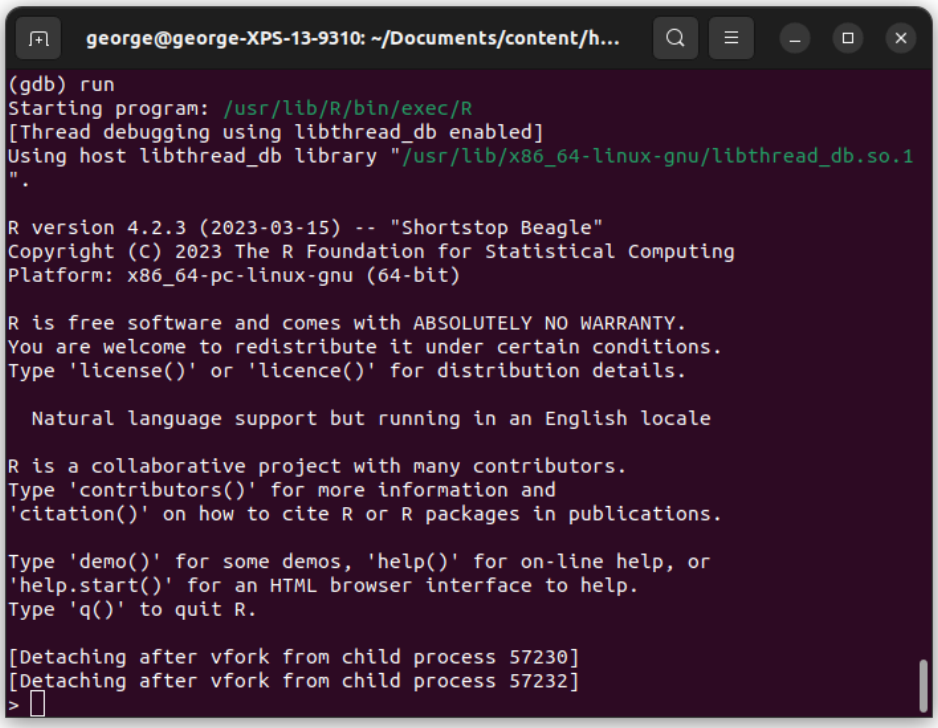
1. Ejecutar R con gdb como depurador: `R --debugger=gdb`. R no iniciará inmediatamente, así que tenemos tiempo para agregar puntos de interrupción.



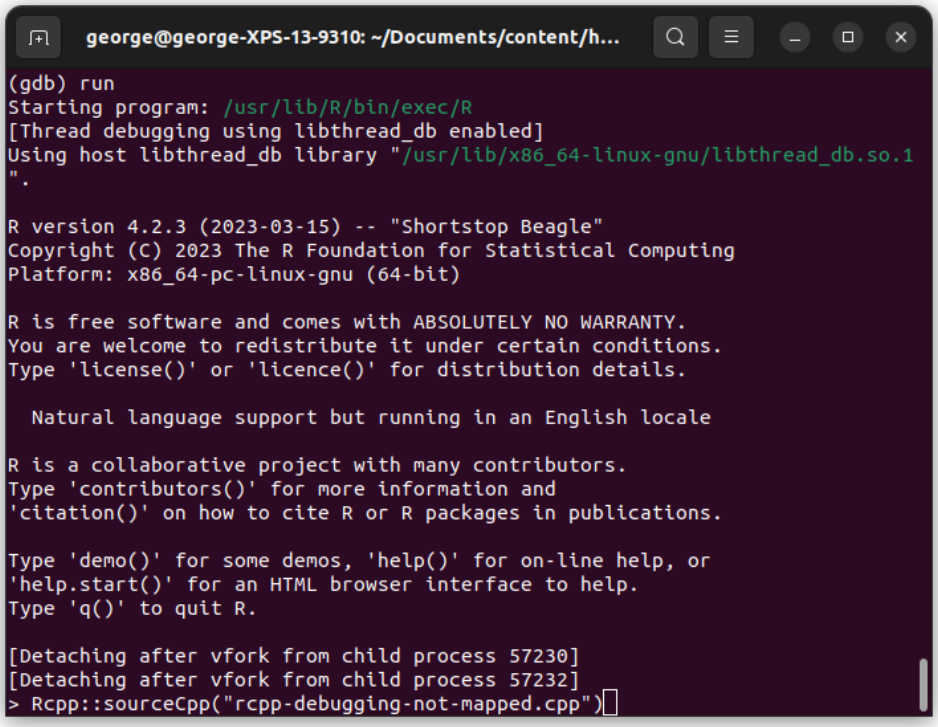
2. Podemos establecer un punto de interrupción en la función dada con `break faulty_program`. GDB lo capturará sobre la marcha, así que elige `yes`. Es muy probable que te advierta que no hay símbolo para esa función.



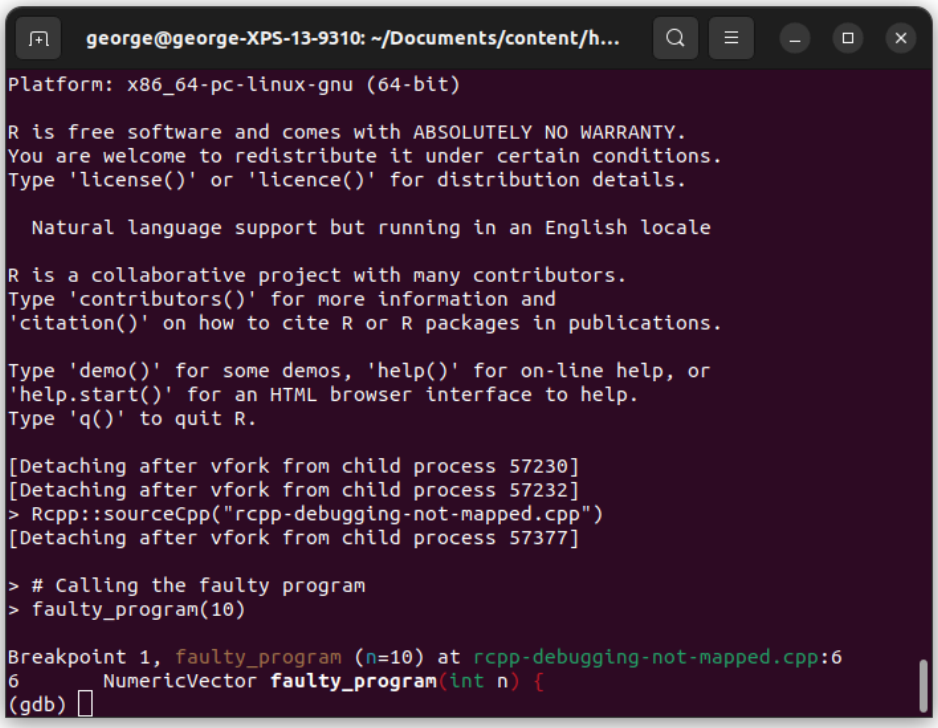
3. Ejecutar R usando el comando `run` en gdb:



4. Obtener el programa usando `Rcpp::sourceCpp`, y esperar a que `gdb` pause el programa una vez que alcance el punto de interrupción.



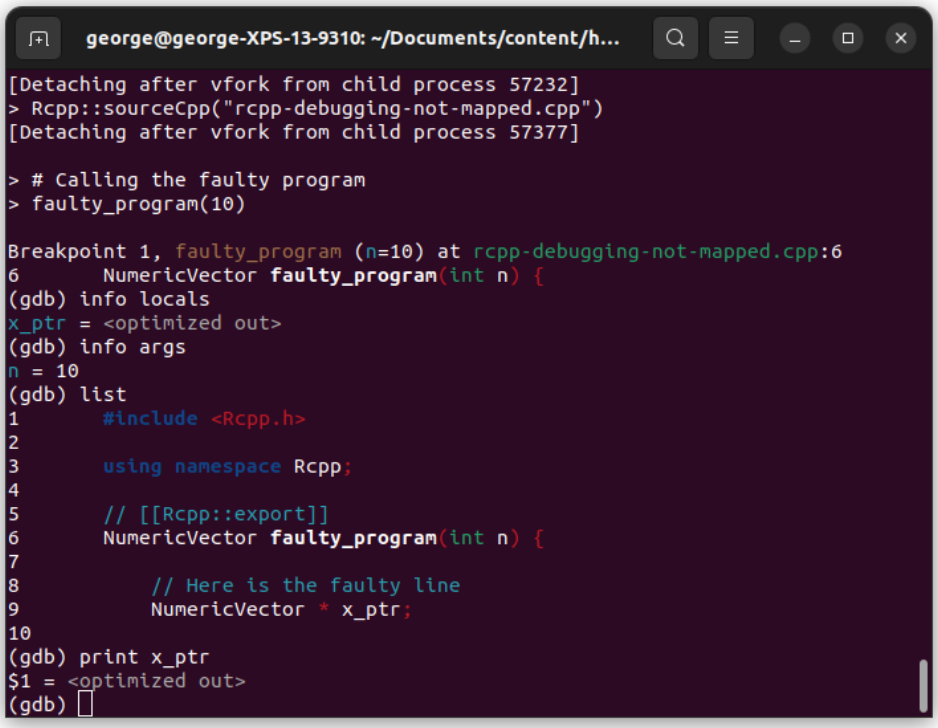
5. Una vez que el programa se ha pausado, podemos inspeccionar el contexto.



Debido al número de opciones que tiene, usar GBD puede ser abrumador. Aquí está la lista de comandos que uso más:

```
help        # Obtener ayuda
info locals # Listar las variables locales (alcance)
info args   # Listar los argumentos pasados a la función
list        # Ver las últimas pocas líneas del código fuente
continue    # Continuar ejecutando el programa
next        # Ejecutar el siguiente paso
bt          # Mostrar toda la pila de llamadas (backtrace)
up          # Subir un nivel en la pila de llamadas
down        # Bajar un nivel en la pila de llamadas
print       # Imprimir/mostrar una expresión
```

Y aquí hay un ejemplo usando `info locals`, `info args`, `list`, y `print`.



6. Finalmente, para salir del programa, escribe `exit` (similar a `q()` en R.)

Appendices

A. Notas sobre alcance en C++

Este capítulo presenta un tema más avanzado pero importante en C++: RAII y alcance (scoping). RAII significa Resource Acquisition Is Initialization, y es un patrón de programación que asegura que los recursos se liberen correctamente cuando ya no se necesitan. Desde la perspectiva de una persona usuaria de R, esto ofrece una buena oportunidad para trabajar con alcance en C++. Veámoslo con un ejemplo:

```
class Agent {
public:
    int fun();
};

// Declaración de la clase Model
class Model {
private:
    Agent agent_;
    int x_;

public:
    inline static Model * instance_ = nullptr;
    virtual int run();
    Model(int x) : agent_(Agent()), x_(x) {};

    // Función clave que permite acceder a `Model`
    // desde llamadas dentro de la pila de ejecución.
    static Model & get();

    int get_x();
};

// Implementación de los métodos de Model
inline int Model::run() {
    Model::instance_ = this;
    return agent_.fun();
};

inline Model & Model::get() {return *instance_;};
inline int Model::get_x() {return x_;};

// Implementación del método de Agent
inline int Agent::fun() {return Model::get().get_x();};

// [[Rcpp::export]]
int test_scoping_cpp() {

    // Crear una instancia de Model
    Model Model(42);

    // Ejecutar el Model
    return Model.run();
}

// [[Rcpp::export]]
bool is_it_on() {
```

```
    return Model::instance_ != nullptr;
}
```

Comprobando si funciona o no:

```
test_scoping_cpp()
```

```
[1] 42
```

¡Funciona! Separamos esto en partes. Primero, la clase `Agent` no tiene acceso explícito a la clase `Model`; en ninguna parte de su definición aparece como miembro. La clave está en la función `static Model::get()`, que puede ser llamada por cualquier función dentro de la pila de ejecución sin tener acceso directo a `Model`:

```
Model::run() -> Agent::fun() -> Model::get() -> Model::get_x()
```

Y la función de `Agent` accede al modelo mediante el miembro estático. Además, podemos ver que después de salir de `Model::run()`, la variable `Model::instance_` sigue siendo no nula:

```
is_it_on()
```

```
[1] TRUE
```

Ahora, esto no es completamente seguro, porque la variable `instance_` seguiría disponible incluso después de llamar a la función `Model::run()`. Para resolverlo, podemos usar una clase auxiliar que restablezca `instance_` a `nullptr`:

```
#include <Rcpp.h>
class Agent {
public:
    int fun();
};

class Scope;

// Declaración de la clase Model
class Model_s {
    friend Scope;
private:
    Agent agent_;
    int x_;

public:
    inline static Model_s * instance_ = nullptr;
    virtual int run();
    Model_s(int x) : agent_(Agent()), x_(x) {};
    static Model_s & get();
    int get_x();
};

// Declaración e implementación del método Scope
class Scope {
private:
    Model_s * prev_;
public:
    explicit Scope(Model_s * model) : prev_(Model_s::instance_) {
        Model_s::instance_ = model;
    };
    ~Scope() {
        Model_s::instance_ = prev_;
    };
};
```

```
// Implementación de los métodos de Model_s
inline int Model_s::run() {

    // Aquí está el punto importante. El constructor
    // de `Scope` fija el valor de `instance_` para
    // que sea la instancia actual de `Model`, pero
    // una vez que salimos de la llamada, el destructor
    // reasigna `instance_` a `nullptr`.
    Scope scope(this);

    return agent_.fun();
};

inline Model_s & Model_s::get() {return *instance_;};
inline int Model_s::get_x() {return x_;};

// Implementación del método de Agent
inline int Agent::fun() {return Model_s::get().get_x();};

// [[Rcpp::export]]
int test_scoping_cpp2() {

    // Crear una instancia de Model
    Model_s Model(42);

    // Ejecutar el Model
    return Model.run();
}

// [[Rcpp::export]]
bool is_it_on2() {
    return Model_s::instance_ != nullptr;
}
```

La clase simple `Scope` ofrece una forma elegante de controlar la duración de esta variable estática:

```
class Scope {
private:
    Model_s * prev_;
public:
    explicit Scope(Model_s * model) : prev_(Model_s::instance_) {
        Model_s::instance_ = model;
    };
    ~Scope() {
        Model_s::instance_ = prev_;
    };
};
```

Al invocarse, establece el valor del miembro estático de `Model_s` durante la vida de `Scope`. Una vez que `Model_s::run()` termina, el destructor de `Scope` restablece el valor de `Model_s::instance_` a su valor previo, `nullptr` en este caso. Aun así, esto permite llamadas anidadas, manteniendo el valor correcto de `instance_` a medida que crece la pila de ejecución. Este es el resultado de la nueva implementación:

```
test_scoping_cpp2() # Verificando que funciona
```

```
[1] 42
```

```
is_it_on2()         # Y que queda apagado
```

```
[1] FALSE
```

> ⚠ Warning
>
> Mientras implementaba esto, estaba usando `Model` en ambos bloques de código C++. Aunque igual compila, el símbolo `Model` se comparte entre ambas implementaciones, lo que rompía mi código (la función `is_it_on2()` devolvía `TRUE` cuando debía devolver `FALSE`).

A.1 Versiones multihilo

Si usas computación paralela con OpenMP y similares, es importante ser cuidadoso. Por defecto, las copias de `Model` comparten el valor de `_instance`, lo que puede no ser deseable. En su lugar, se recomienda usar la palabra clave `thread_local`:

```
class Model {
    static thread_local instance_;
}

// Debe inicializarse fuera de la clase
thread_local Model::instance_ = nullptr;
```

Esto asegura que, si creas copias de `Model`, cada hilo tenga acceso a su propia copia.

A.2 Trabajando con múltiples versiones compiladas

Otra advertencia que he experimentado ocurre cuando hay dos objetos de biblioteca C++ diferentes (lo que Rcpp crea como `.so`, `.o` o `.dll`, según el sistema operativo). En particular, observé esto durante el desarrollo de los paquetes `epiworldR` y `measles` en R. Ambos tenían bibliotecas compiladas que dependían de la biblioteca de plantillas C++ `epiworld`, así que al cargarlos juntos, el acceso mediante miembro estático no funcionaba como se esperaba; en esos casos, es mejor ser más explícito y simplemente pasar `Model` como otro argumento; en otras palabras, no intentes ser "demasiado inteligente" y usa algo más confiable.

B. Misceláneos

⚠ Nota de Traducción

Esta versión del capítulo fue traducida de manera automática utilizando IA. El capítulo aún no ha sido revisado por un humano.

B.1 Recursos generales

El Centro de Computación de Investigación Avanzada (anteriormente HPCC) tiene toneladas de recursos en línea. Aquí hay un par de enlaces útiles:

- **Sitio web del Centro de Computación de Investigación Avanzada** https://carc.usc.edu
- **Foro de usuarios (¡muy útil!)** https://hpc-discourse.usc.edu/categories
- **Monitorea tu cuenta** https://hpcaccount.usc.edu/
- **Plantillas de trabajos Slurm** https://carc.usc.edu/user-information/user-guides/high-performance-computing/slurm-templates
- **Usando R** https://carc.usc.edu/user-information/user-guides/software-and-programming/r

B.2 Punteros de datos

En mi humilde opinión, estas son las cosas más importantes que debes saber sobre la gestión de datos en el HPC de USC:

1. Haz tu transferencia de datos usando los nodos de transferencia (es más rápido).
2. Nunca uses tu directorio home como espacio de almacenamiento (usa en su lugar el espacio asignado de tu proyecto).
3. Usa el sistema de archivos scratch solo para datos temporales, es decir, nunca guardes archivos importantes en scratch.
4. Finalmente, además del **Protocolo de copia segura (scp)**, si eres como yo, trata de configurar un cliente GUI para mover tus datos (ver esto).

B.3 Las opciones de Slurm que se olvidaron de contarte...

Primero que nada, tienes que estar consciente de que lo único que hace Slurm es asignar recursos. Si tu aplicación usa computación paralela o no, esa es otra historia.

Aquí algunas opciones de las que necesitas estar consciente:

- `ntasks` (predeterminado 1) Esto le dice a Slurm cuántos procesos tendrás ejecutándose. Ten en cuenta que los procesos no necesitan estar en el mismo nodo (así que Slurm puede reservar espacio en múltiples nodos)
- `cpus-per-task` (predeterminado 1) Esto es cuántos CPUs cada tarea estará usando. Esto es lo que necesitas usar si estás usando OpenMP (o un paquete que use eso), o cualquier cosa que necesites mantener dentro del mismo nodo.
- `nodes` el número de nodos que quieres usar en tu trabajo. Esto es útil principalmente si te importa el número máximo (yo diría) de nodos que quieres que use tu trabajo. Entonces, por ejemplo, si quieres usar 8 núcleos para una sola tarea y forzarla a estar en el mismo nodo, agregarías la opción `--nodes=1/1`.

- `mem-per-cpu` (predeterminado 1GB) Esta es la cantidad MÍNIMA de memoria que quieres que Slurm asigne para la tarea. No es una barrera rígida, así que tu proceso puede ir por encima de eso.
- `time` (predeterminado 30min) Este también es un límite rígido, así que si tu trabajo toma más que el tiempo especificado, Slurm lo matará.
- `partition` (predeterminado ““) y `account` (predeterminado”“) estas dos opciones van juntas, esto le dice a Slurm qué recursos usar. Además de los recursos privados tenemos los siguientes:
  - **partición quick**: Cualquier trabajo que sea lo suficientemente pequeño (en términos de tiempo y memoria) irá por este camino. Este es usualmente el predeterminado si no especificas ninguna opción de memoria o tiempo.
  - **partición main**: Los trabajos que requieren más recursos irán en esta línea.
  - **partición scavenge**: Si necesitas un número masivo de recursos, y tienes un trabajo que no debería, en principio, tomar demasiado tiempo para finalizar (menos de un par de horas), y **estás de acuerdo con que alguien lo mate**, entonces esta cola es para ti. La partición Scavenge usa todos los recursos inactivos de las particiones privadas, así que si cualquiera de los propietarios solicita los recursos, Slurm cancelará tu trabajo, es decir, no tienes prioridad (ver más).
  - **partición largemem**: Si necesitas mucha memoria, tenemos 4 nodos de 1TB para eso.

  Más información sobre las particiones aquí

B.4 Buenas prácticas (recomendaciones)

Esto es lo que deberías usar como mínimo:

```
#SBATCH --output=simulation.out
#SBATCH --job-name=simulation
#SBATCH --time=04:00:00
#SBATCH --mail-user=[tu]@usc.edu
#SBATCH --mail-type=END,FAIL
```

- `output` es el nombre del archivo de registro al que Slurm escribirá.
- `job-name` es eso, el nombre del trabajo. Puedes usar esto para matar o al menos ser capaz de identificar qué es lo que estás ejecutando cuando usas `myqueue`
- `time` Trata siempre de establecer una estimación de tiempo (más un poco más) para tu trabajo.
- `mail-user`, `mail-type` para que Slurm te notifique cuando las cosas sucedan

También, en tu código R

- Cualquier I/O debería hacerse a Scratch (`/scratch/[tu id de red usc]`) o Tmp `Sys.getenv("TMPDIR")`.

B.5 Ejecutando R interactivamente

1. El HPC tiene varias piezas de software preinstaladas. R es una de esas.
2. Para acceder al software preinstalado, usamos el **sistema de módulos Lmod** (más información **aquí**)
3. Tiene múltiples versiones de R instaladas. Usa tu favorita ejecutando

   ```
   module load R/4.2.2/[número de versión]
   ```

   Donde `[número de versión]` puede ser 3.5.6 y hasta 4.0.3 (la última actualización). El módulo `usc` automáticamente carga gcc/8.3.0, openblas/0.3.8, openmpi/4.0.2, y pmix/3.1.3.
4. Nunca es una buena idea usar tu directorio home para instalar paquetes R, por eso deberías tratar de usar un **enlace simbólico en su lugar**, así

   ```
   cd ~
   mkdir -p /ruta/a/un/proyecto/con/mucho/espacio/R
   ln -s /ruta/a/un/proyecto/con/mucho/espacio/R R
   ```

De esta manera, cada vez que instales tus paquetes R, R tendrá como predeterminado esa ubicación

5. Puedes ejecutar sesiones interactivas en HPC, pero esto se recomienda hacerlo usando la función `salloc` en Slurm, en otras palabras, ¡NUNCA USES R (O CUALQUIER SOFTWARE) PARA HACER ANÁLISIS DE DATOS EN LOS NODOS CABEZA! Las opciones pasadas a salloc son las mismas opciones que pueden pasarse a `sbatch` (ver la siguiente sección). Por ejemplo, si necesito hacer algunos análisis en la partición `thomas` (que es privada y tengo acceso a ella), escribiría algo como

```
salloc --account=lc_pdt --partition=thomas --time=02:00:00 --mem-per-cpu=2G
```

Esto me pondría en un solo nodo asignando 2 gigs de memoria por un máximo de 2 horas.

B.6 NoNos cuando uses R

- Hacer cómputo en el nodo cabeza (compilar cosas está bien)
- Solicitar un número de nodos (a menos que sepas lo que estás haciendo)
- Usar tu directorio home para I/O
- Guardar información importante en Staging/Scratch

C. Referencias

⚠ Nota de Traducción

Esta versión del capítulo fue traducida de manera automática utilizando IA. El capítulo aún no ha sido revisado por un humano.

Referencias

C.1 Recursos Adicionales

C.1.1 Libros Recomendados

- **Advanced R** por Hadley Wickham
- **Efficient R Programming** por Colin Gillespie y Robin Lovelace
- **Parallel R** por Q. Ethan McCallum y Stephen Weston
- **Seamless R and C++ Integration with Rcpp** por Dirk Eddelbuettel

C.1.2 Sitios Web y Documentación

- CRAN Task View: High-Performance Computing
- Rcpp Documentation
- SLURM Documentation
- R Parallel Computing

C.1.3 Paquetes Relevantes

- `parallel`: Computación paralela básica
- `foreach`: Bucles paralelos
- `future`: Framework unificado para computación paralela
- `Rcpp`: Integración R/C++
- `data.table`: Procesamiento eficiente de datos
- `bench`: Benchmarking de código

D. Novedades

D.1 Versión 2026.03.18

- Se agregó un nuevo ejemplo en el capítulo del paquete `parallel` que muestra cómo reemplazar un for-loop con `parLapply()`, incluyendo una discusión sobre semillas por simulación como alternativa a `clusterSetRNGStream`.

D.2 Versión 2025.09.03

- Desde ahora, el libro cuenta con una sección de novedades. Intentaré mantener esta al día incluyendo los cambios realizados entre versiones. Versiones anteriores se encontrarán disponibles en GitHub como "releases".

Bibliography

[1] R Core Team, "R: A Language and Environment for Statistical Computing." Vienna, Austria, 2023. [Online]. Available: https://www.r-project.org/

[2] M. Dowle and A. Srinivasan, "data.table: Extension of `data.frame`." 2021. [Online]. Available: https://cran.r-project.org/package=data.table

[3] LUMI consortium, "Documentation - Distribution and binding." [Online]. Available: https://docs.lumi-supercomputer.eu/runjobs/scheduled-jobs/distribution-binding/

[4] A. B. Yoo, M. A. Jette, and M. Grondona, "SLURM: Simple Linux Utility for Resource Management," in *Job Scheduling Strategies for Parallel Processing*, vol. 2862, D. Feitelson, L. Rudolph, and U. Schwiegelshohn, Eds., in Lecture Notes in Computer Science, vol. 2862., Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 44–60. doi: 10.1007/10968987_3.

[5] G. Vega Yon and P. Marjoram, "slurmR: A lightweight wrapper for HPC with Slurm," *The Journal of Open Source Software*, vol. 4, no. 39, July 2019, doi: 10.21105/joss.01493.

[6] G. Vega Yon and P. Marjoram, "slurmR: A Lightweight Wrapper for 'Slurm'." 2022. [Online]. Available: https://github.com/USCbiostats/slurmR

[7] D. Eddelbuettel and R. François, "Rcpp: Seamless R and C++ Integration," *Journal of Statistical Software*, vol. 40, no. 8, pp. 1–18, 2011, doi: 10.18637/jss.v040.i08.

[8] D. Eddelbuettel, *Seamless R and C++ Integration with Rcpp*. New York: Springer, 2013. doi: 10.1007/978-1-4614-6868-4.

[9] D. Eddelbuettel and J. J. Balamuta, "Extending extitR with extitC++: A Brief Introduction to extitRcpp," *The American Statistician*, vol. 72, no. 1, pp. 28–36, 2018, doi: 10.1080/00031305.2017.1375990.

[10] K. Rue-Albrecht, D. Cassol, J. Rainer, and L. Shepherd, *Welcome | Bioconductor Packages: Development, Maintenance, and Peer Review*.