Stat 3701 Lecture Notes: Parallel Computing in R

Charles J. Geyer

August 12, 2020

1 License

2 Note

These notes have not been kept up to date. The best version of my class notes for parallel computing are those for Stat 8054 (PhD level statistical computing). Those notes say more or less the same as these but have many corrections (the code for LATIS actually works). They are, however, terser.

Also the section about the compiler below should probably be ignored. It was written before the just-in-time compiler was turned on by default in R version 3.4.0. Now all code whether precompiled or not should get most of the speed-up possible with no special action taken by users. There is nothing wrong with what the section below says; it just isn't necessary anymore.

3 R

- The version of R used to make this document is 4.0.2.
- The version of the `rmarkdown` package used to make this document is 2.3.

4 Computer History

The first computer I ever worked on was an IBM 1130 (Wikipedia page). This was in 1971. It was the only computer the small liberal arts college I went to had.

It had 16 bit words and 32,768 of them (64 kilobytes) of memory. The clock speed was 278 kHz (kilohertz).

For comparison, the laptop I am working on has 64 bit words and 8054820 kB (8 GB, which is 125,000 times as much as the IBM 1130). Its clock speed is 2.50GHz (8,993 times as fast).

That is `log2(8993)` = 13.13 doublings of speed in 46 years, which is a doubling of computer speed every 3.5 years.

For a very long time (going back to even before 1970), computers have been getting faster at more or less this rate, but something happened in 2003 as discussed in the article The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software.

This exponential growth in computer speed was often confused with Moore's Law (Wikipedia article), which actually predicts a doubling of the *number of transistors* on a computer chip every 2 years. For a long time this went hand in hand with a doubling in computer speed, but about 2003 the speed increases stopped while the number of transistors kept doubling.

What to do with all of those transistors? Make more processors. So even the dumbest chips today, say the one in your phone, has multiple "cores", each a real computer. High end graphics cards, so-called GPU's, can run thousands of processes simultaneously, but have a very limited instruction set. They can only run specialized graphics code very fast. They could not run 1000 instances of R very fast. Your laptop, in contrast, can run multiple instances of R very fast, just not so many. The laptop I use for class has an Intel i5 chip with 4 cores that can execute 4 separate processes at full machine speed. The desktop in my office has an Intel i7 chip that has 4 cores each of which can handle 2 "hyperthreads" so it can execute 8 separate processes at (nearly) full machine speed. Big servers can handle even more parallel processes. Compute clusters like at the U of M supercomputer center or at CLA research computing can handle even more parallel processes.

In summary, you get faster today by running more processes in parallel not by running faster on one processor.

5 Task View

The task view on high performance computing includes discussion of parallel processing (since that is what high performance computing is all about these days).

But, somewhat crazily, the task view does not discuss the most important R package of all for parallel computing. That is R package `parallel` in the R base (the part of R that must be installed in each R installation).

This is the only R package for high performance computing that we are going to use in this course.

6 An Example

6.1 Introduction

The example that we will use throughout this document is simulating the sampling distribution of the MLE for Normal(θ, θ^2) data.

This is the same as the example of Section 5.3 of the course notes on simulation, except we are going to simplify the R function `estimator` using some ideas from later on in the notes (Section 6.2.4 of the course notes on the bootstrap).

6.2 Set-Up

```
n <- 10
nsim <- 1e4
theta <- 1

doit <- function(estimator, seed = 42) {
    set.seed(seed)
    result <- double(nsim)
    for (i in 1:nsim) {
        x <- rnorm(n, theta, abs(theta))
        result[i] <- estimator(x)
    }
```

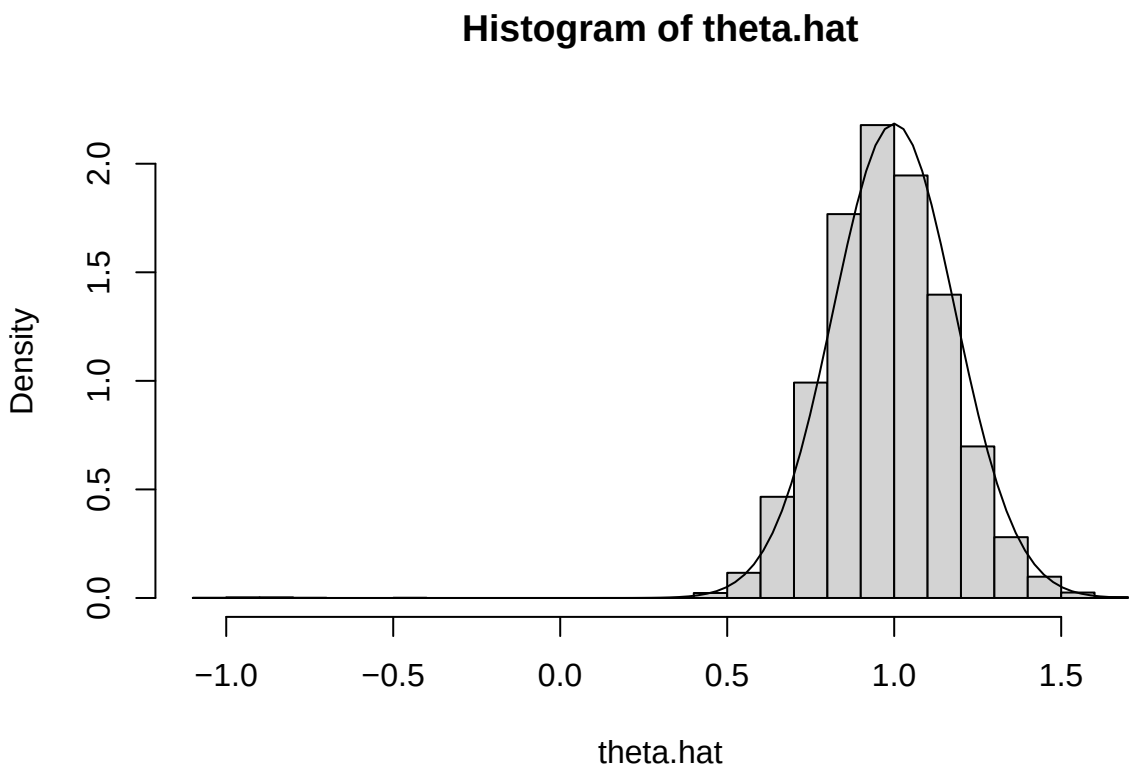
```
    return(result)
}

mlogl <- function(theta, x) sum(- dnorm(x, theta, abs(theta), log = TRUE))

mle <- function(x) {
    if (all(x == 0))
        return(0)
    nout <- nlm(mlogl, sign(mean(x)) * sd(x), x = x)
    while (nout$code > 3)
        nout <- nlm(mlogl, nout$estimate, x = x)
    return(nout$estimate)
}
```

6.3 Try It

```
theta.hat <- doit(mle)
```

6.4 Check It

```
hist(theta.hat, probability = TRUE, breaks = 30)
curve(dnorm(x, mean = theta, sd = theta / sqrt(3 * n)), add = TRUE)
```



The curve is the PDF of the asymptotic normal distribution of the MLE, which uses the formula

$$I_n(\theta) = \frac{3n}{\theta^2}$$

which isn't in these course notes (although we did calculate Fisher information for any given numerical value of θ in the practice problems solutions cited above).

Looks pretty good. The large negative estimates are probably not a mistake. The parameter is allowed to be negative, so sometimes the estimates come out negative even though the truth is positive. And not just a little negative because $|\theta|$ is also the standard deviation, so it cannot be small and the model fit the data.

6.5 Time It

Now for something new. We will time it.

```
time1 <- system.time(theta.hat.mle <- doit(mle))
time1
```

```
##    user  system elapsed
##   0.764   0.000   0.763
```

6.6 Time It More Accurately

That's too short a time for accurate timing. Also we should probably average over several IID iterations to get a good average. Try again.

```
nsim <- 1e5
nrep <- 7
time1 <- NULL
for (irep in 1:nrep)
    time1 <- rbind(time1, system.time(theta.hat.mle <- doit(mle)))
time1
```

```
##      user.self sys.self elapsed user.child sys.child
## [1,]     8.028    0.000   8.036          0         0
## [2,]     7.976    0.000   7.990          0         0
## [3,]     7.920    0.000   7.924          0         0
## [4,]     7.872    0.000   7.882          0         0
## [5,]     7.912    0.000   7.925          0         0
## [6,]     7.880    0.004   7.901          0         0
## [7,]     8.016    0.000   8.022          0         0
```

```
apply(time1, 2, mean)
```

```
##    user.self     sys.self      elapsed   user.child    sys.child
## 7.9434285714 0.0005714286 7.9542857143 0.0000000000 0.0000000000
```

```
apply(time1, 2, sd) / sqrt(nrep)
```

```
##    user.self     sys.self      elapsed   user.child    sys.child
## 0.0239750437 0.0005714286 0.0230823725 0.0000000000 0.0000000000
```

7 R Package `compiler`

7.1 The Compiler

Also very important for speed is the R package `compiler` which is also in the R base (the part of R that must be installed in each R installation).

Compiling R is not like compiling C, C++, or Java or other so-called "compiled" languages. Compiled R works just the same as non-compiled R. You don't even notice it. The only difference is that it runs approximately twice as fast.

Since version 2.14.0 of R (31 Oct 2011) all base and recommended packages are compiled during the installation process. Any code you write yourself is not compiled unless you do it explicitly, as in Section 6.4 below.

Since version 3.4.0 of R (20 Apr 2017) the just-in-time compiler (JIT) is turned on by default at its highest level. This means

- larger closures (functions) are compiled before their first use,
- some small closures are also compiled before their second use, and
- all top level loops are compiled before they are executed.

And all of this happens automatically with nothing being done by the user except using R as always. So using versions 3.4.0 or higher may make explicit use of the compiler unnecessary.

7.2 Compiling Our Functions

Now compile all of our functions and see if this helps.

```
library(compiler)
mlogl
```

```
## function(theta, x) sum(- dnorm(x, theta, abs(theta), log = TRUE))
## <bytecode: 0x559ab47d16a0>
```

```
mlogl <- cmpfun(mlogl)
mlogl
```

```
## function(theta, x) sum(- dnorm(x, theta, abs(theta), log = TRUE))
## <bytecode: 0x559ab4fd82e0>
```

```
mlogl(1.4, rnorm(6))
```

```
## [1] 11.83809
```

After compilation functions look the same except for extra blurfle about byte code. And they work the same.

So compile the rest.

```
doit <- cmpfun(doit)
mle <- cmpfun(mle)
```

7.3 Example With Compilation

The following code chunk is identical to the code chunk in Section 6.2 above except for changing `time1` to `time2` everywhere.

```
time2 <- NULL
for (irep in 1:nrep)
    time2 <- rbind(time2, system.time(theta.hat.mle <- doit(mle)))
time2
```

```
##      user.self sys.self elapsed user.child sys.child
## [1,]     7.716    0.004   7.729          0         0
## [2,]     7.684    0.016   7.728          0         0
## [3,]     7.688    0.008   7.701          0         0
```

```
## [4,]     7.880    0.000   7.888          0         0
## [5,]     7.604    0.000   7.612          0         0
## [6,]     7.712    0.000   7.720          0         0
## [7,]     7.708    0.000   7.720          0         0
```

```
apply(time2, 2, mean)
```

```
##  user.self   sys.self    elapsed user.child  sys.child
##   7.713143   0.004000   7.728286   0.000000   0.000000
```

```
apply(time2, 2, sd) / sqrt(nrep)
```

```
##   user.self    sys.self     elapsed  user.child   sys.child
## 0.031347080 0.002309401 0.030827256 0.000000000 0.000000000
```

It is actually slower when compiled! But not statistically significantly so.

```
t.test(time1[ , 1], time2[ , 1])
```

```
##
##  Welch Two Sample t-test
##
## data:  time1[, 1] and time2[, 1]
## t = 5.8353, df = 11.23, p-value = 0.0001044
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  0.1436416 0.3169299
## sample estimates:
## mean of x mean of y
##  7.943429  7.713143
```

What happened? Most of the time is spent in the R function `nlm` which is mostly C code. So it has always run at C speed (as fast as the computer can go) even before R had a compiler. So that means there is not much for the compiler to do. Or so I hypothesize.

7.4 Profiling the Example

We should have done this first (see also Section 6.5 below).

```
time3 <- NULL
Rprof()
for (irep in 1:nrep)
    time3 <- rbind(time3, system.time(theta.hat.mle <- doit(mle)))
Rprof(NULL)
time3
```

```
##      user.self sys.self elapsed user.child sys.child
## [1,]     7.932    0.004   7.955          0         0
## [2,]     7.804    0.000   7.806          0         0
## [3,]     7.756    0.000   7.762          0         0
## [4,]     7.648    0.000   7.654          0         0
## [5,]     7.716    0.000   7.721          0         0
## [6,]     7.816    0.000   7.820          0         0
## [7,]     7.788    0.000   7.792          0         0
```

```
apply(time3, 2, mean)
```

```
##    user.self     sys.self      elapsed   user.child    sys.child
```

```
## 7.7800000000 0.0005714286 7.7871428571 0.0000000000 0.0000000000
```

```
apply(time3, 2, sd) / sqrt(nrep)
```

```
##    user.self     sys.self      elapsed   user.child    sys.child
## 0.0335005330 0.0005714286 0.0353111534 0.0000000000 0.0000000000
```

```
summaryRprof()
```

```
## $by.self
##                 self.time self.pct total.time total.pct
## "dnorm"             15.46    28.25      15.46     28.25
## "f"                  8.82    16.12      24.28     44.37
## ".External2"         4.96     9.06      33.68     61.55
## "<Anonymous>"        4.44     8.11      28.72     52.49
## "nlm"                3.50     6.40      50.76     92.76
## "rnorm"              2.24     4.09       2.24      4.09
## "match.call"         1.92     3.51       4.76      8.70
## "stopifnot"          1.90     3.47       7.06     12.90
## "sys.parent"         1.46     2.67       1.46      2.67
## "mean"               1.36     2.49       2.48      4.53
## "var"                1.28     2.34      10.08     18.42
## "estimator"          1.00     1.83      51.76     94.59
## "is.data.frame"      0.98     1.79       0.98      1.79
## "mean.default"       0.92     1.68       1.12      2.05
## "pmatch"             0.74     1.35       0.76      1.39
## "sys.call"           0.56     1.02       1.04      1.90
## "doit"               0.54     0.99      54.54     99.67
## "sys.function"       0.42     0.77       1.40      2.56
## "parent.frame"       0.40     0.73       0.40      0.73
## "max"                0.28     0.51       0.28      0.51
## "all"                0.22     0.40       0.22      0.40
## "gc"                 0.18     0.33       0.18      0.33
## "length"             0.18     0.33       0.18      0.33
## "sum"                0.18     0.33       0.18      0.33
## "sd"                 0.12     0.22      10.20     18.64
## "/"                  0.12     0.22       0.12      0.22
## "^"                  0.12     0.22       0.12      0.22
## "is.numeric"         0.10     0.18       0.10      0.18
## "sqrt"               0.08     0.15       0.08      0.15
## "anyNA"              0.06     0.11       0.06      0.11
## "as.integer"         0.06     0.11       0.06      0.11
## "*"                  0.04     0.07       0.04      0.07
## "is.atomic"          0.04     0.07       0.04      0.07
## "as.character"       0.02     0.04       0.02      0.04
## "invisible"          0.02     0.04       0.02      0.04
##
## $by.total
##                      total.time total.pct self.time self.pct
## "block_exec"              54.72    100.00      0.00     0.00
## "call_block"              54.72    100.00      0.00     0.00
## "eval"                    54.72    100.00      0.00     0.00
## "evaluate_call"           54.72    100.00      0.00     0.00
## "evaluate"                54.72    100.00      0.00     0.00
## "handle"                  54.72    100.00      0.00     0.00
```

```
## "in_dir"                  54.72    100.00      0.00     0.00
## "knitr::knit"             54.72    100.00      0.00     0.00
## "process_file"            54.72    100.00      0.00     0.00
## "process_group.block"     54.72    100.00      0.00     0.00
## "process_group"           54.72    100.00      0.00     0.00
## "rbind"                   54.72    100.00      0.00     0.00
## "render"                  54.72    100.00      0.00     0.00
## "system.time"             54.72    100.00      0.00     0.00
## "timing_fn"               54.72    100.00      0.00     0.00
## "withCallingHandlers"     54.72    100.00      0.00     0.00
## "withVisible"             54.72    100.00      0.00     0.00
## "doit"                    54.54     99.67      0.54     0.99
## "estimator"               51.76     94.59      1.00     1.83
## "nlm"                     50.76     92.76      3.50     6.40
## ".External2"              33.68     61.55      4.96     9.06
## "<Anonymous>"             28.72     52.49      4.44     8.11
## "f"                       24.28     44.37      8.82    16.12
## "dnorm"                   15.46     28.25     15.46    28.25
## "sd"                      10.20     18.64      0.12     0.22
## "var"                     10.08     18.42      1.28     2.34
## "stopifnot"                7.06     12.90      1.90     3.47
## "match.call"               4.76      8.70      1.92     3.51
## "mean"                     2.48      4.53      1.36     2.49
## "rnorm"                    2.24      4.09      2.24     4.09
## "sys.parent"               1.46      2.67      1.46     2.67
## "sys.function"             1.40      2.56      0.42     0.77
## "mean.default"             1.12      2.05      0.92     1.68
## "sys.call"                 1.04      1.90      0.56     1.02
## "is.data.frame"            0.98      1.79      0.98     1.79
## "pmatch"                   0.76      1.39      0.74     1.35
## "parent.frame"             0.40      0.73      0.40     0.73
## "max"                      0.28      0.51      0.28     0.51
## "all"                      0.22      0.40      0.22     0.40
## "gc"                       0.18      0.33      0.18     0.33
## "length"                   0.18      0.33      0.18     0.33
## "sum"                      0.18      0.33      0.18     0.33
## "/"                        0.12      0.22      0.12     0.22
## "^"                        0.12      0.22      0.12     0.22
## "is.numeric"               0.10      0.18      0.10     0.18
## "sqrt"                     0.08      0.15      0.08     0.15
## "anyNA"                    0.06      0.11      0.06     0.11
## "as.integer"               0.06      0.11      0.06     0.11
## "*"                        0.04      0.07      0.04     0.07
## "is.atomic"                0.04      0.07      0.04     0.07
## "as.character"             0.02      0.04      0.02     0.04
## "invisible"                0.02      0.04      0.02     0.04
##
## $sample.interval
## [1] 0.02
##
## $sampling.time
## [1] 54.72
```

This is a little hard to read. Only the first table (the `by.self` component of the output) is important. The

row labels are function names, some of which we recognize like

```
dnorm
nlm
rnorm
mean
estimator
doit
```

and some we do not recognize like

```
f
<Anonymous>
.External2
mean.default
```

So our first task is to understand the ones we do not recognize. To do that we need to be very fussy about our language, distinguishing between R objects and *names* of R objects. We think of `mlogl` as an R function, but it is not. It is the *name* of an R function that can have various names.

```
fred <- sally <- herman <- alice <- mlogl
```

Now it has five different names, but there is only one R object that those five names refer to. When a function is an argument to another function (which makes the latter a "higher-order function") it gets a new name inside the latter. That's how function arguments work for all kinds of objects.

```
args(nlm)
```

```
## function (f, p, ..., hessian = FALSE, typsize = rep(1, length(p)),
##     fscale = 1, print.level = 0, ndigit = 12, gradtol = 1e-06,
##     stepmax = max(1000 * sqrt(sum((p/typsize)^2)), 1000), steptol = 1e-06,
##     iterlim = 100, check.analyticals = TRUE)
## NULL
```

The first argument of `nlm` is called `f`. So when we call `nlm` with its first argument being `mlogl`, the function that is *named* `mlogl` outside of `nlm` is *named* `f` inside `nlm`. So now we understand what `f` is. It is the same R object as `mlogl`. When this function is called over and over again inside `mlogl` (because it needs to evaluate the objective function at many points to minimize it), it is named `f` because that is what the first argument of `nlm` is *named*.

To explain some of the rest we need to look at all of the code of `nlm`

```
nlm
```

```
## function (f, p, ..., hessian = FALSE, typsize = rep(1, length(p)),
##     fscale = 1, print.level = 0, ndigit = 12, gradtol = 1e-06,
##     stepmax = max(1000 * sqrt(sum((p/typsize)^2)), 1000), steptol = 1e-06,
##     iterlim = 100, check.analyticals = TRUE)
## {
##     print.level <- as.integer(print.level)
##     if (print.level < 0 || print.level > 2)
##         stop("'print.level' must be in {0,1,2}")
##     msg <- (1 + c(8, 0, 16))[1 + print.level]
##     if (!check.analyticals)
##         msg <- msg + (2 + 4)
##     .External2(C_nlm, function(x) f(x, ...), p, hessian, typsize,
##         fscale, msg, ndigit, gradtol, stepmax, steptol, iterlim)
## }
## <bytecode: 0x559ab6657618>
```

```
## <environment: namespace:stats>
```

There we see `.External2` which the documentation `help(".External2")` is clear as mud. It says that `.External2` is just like `.External` but fancier. So we look at `help(".External")` the "Description" of which says

> Functions to pass R objects to compiled C/C++ code that has been loaded into R.

So we see this call to `.External2` is the end of R and the start of C or C++ (and when that C or C++ function returns, it returns to R). So there is nothing the R compiler can do to speed up `.External2`. Only the C or C++ compiler can speed that that up (by using higher optimization level or a newer version of the compiler with more tricks).

The first argument of `.External2` is an R symbol (also called *name*) that refers to the C or C++ function being called. The second argument is an anonymous function

```
function(x) f(x, ...)
```

the point of which is to capture the `...` arguments so the C or C++ code does not need to deal with them. When the function *named* `f` (also *named* `mlogl`) is called, it is called with the `...` arguments passed to it. When we call `nlm` we call it as follows

```
nlm(mlogl, mean(x), x = x)
```

with two *positional arguments, which match the first two arguments of* ***`nlm`*** *(called* ***`f`*** *and* ***`p`*** *inside* ***`nlm`****) and one named argument* named* `x`. Since no argument *name* of `nlm` that comes before `...` starts with `x` and no argument that comes after `...` is exactly `x` (arguments after `...` must be matched exactly), our `x` becomes a `...` argument *named* `x`. And then when the *anonymous* function

```
function(x) f(x, ...)
```

is called, it is has only one argument (also called `x`) that is passed as a *positional* argument to `f` and so matches the first argument to `f` which is

```
args(mlogl)
```

```
## function (theta, x)
## NULL
```

`theta` (so what is called `x` in the anonymous function is called `theta` in the function it calls) and that leaves the second argument which is a *named* argument *named* `x`. So everything works.

All of that was, no doubt, *way more that you wanted to know* but necessary to disentangle what is going on.

Here is what happens.

- We call `nlm`.
- It calls `.External2`.
- Now we are in C or C++ rather than in R. Every time this C or C++ code needs to evaluate the objective function, it evaluates the R function which was the second argument to `.External2`, which is the *anonymous function*

  ```
  function(x) f(x, ...)
  ```

  which calls the function that it names `f` but we think of as `mlogl`.

All of the above was touched on in Section 7 of the course notes on "basics", but we see it can get very messy when one tries to figure out *exactly* what is going on even in fairly simple code.

Finally, `mean` is a generic function and `mean.default` is one of its methods, the one that is used for objects that no other method of `mean` handles. Generic functions, we haven't really covered yet. (Now there are course notes on that subject.) Suffice it to say that `mean.default` is the *name* of a function that is called when we call `mean`.

All of that just to explain the names! Then we want to look at the column of the table labeled `self.pct` which is the percentage of the computer time taken by the function itself as opposed to other functions it calls.

So IMHO the only part of the output worth looking at is summarized in the following table

function name	self.pct
`dnorm`	28.37
`f` (a. k. a. `mlogl`)	16.47
`.External2`	9.02
anonymous function that calls `f`	6.45
`nlm`	6.66
`rnorm`	3.42
`mean`	3.03
`estimator`	2.3

So 28.37% of the computing time is taken up by, `dnorm`, which the compiler cannot speed up because `.Call` is (if we look at its help) another way to go from R to C or C++. So `dnorm` is almost entirely C code, which the R compiler cannot speed up. Then another 9.02% of the computing time is is taken by `.External2` which the R compiler also cannot speed up.

Since the anonymous function that calls `f` a. k. a. `mlogl` does almost nothing, it cannot be speeded up much. I don't understand and cannot explain how it takes up 6.45% of the computer time when there is almost nothing there. Our compiling could conceivably speed up `f` (a. k. a. `mlogl`) `estimator` (a. k. a. `mle`) and `doit` but they take hardly any of the computing time. The only part we could actually (possibly) speed up by writing better R would be the time spent in `f`, which takes up only 16.47% of the computer time.

7.5 The Three Rules of Optimization

Actually, we should have read this first: the Three Rules of Optimization.

8 Parallel Computing

8.1 With Unix Fork and Exec

This method is by far the simplest but

- it only works on one computer (using however many simultaneous processes the computer can do), and
- it does not work on Windows.

First a toy problem that does nothing except show that we are actually using different processes.

```
library(parallel)
ncores <- detectCores()
mclapply(1:ncores, function(x) Sys.getpid(), mc.cores = ncores)
```

```
## [[1]]
## [1] 13490
##
## [[2]]
## [1] 13491
##
```

```
## [[3]]
## [1] 13492
##
## [[4]]
## [1] 13493
##
## [[5]]
## [1] 13494
##
## [[6]]
## [1] 13495
##
## [[7]]
## [1] 13496
##
## [[8]]
## [1] 13497
```

8.1.1 Parallel Streams of Random Numbers

8.1.1.1 Try 1

If we generate random numbers reproducibly, it does not work using the default RNG.

```
set.seed(42)
mclapply(1:ncores, function(x) rnorm(5), mc.cores = ncores)
```

```
## [[1]]
## [1]  0.64670759 -0.94958318  0.06697775  1.16957011  1.10814651
##
## [[2]]
## [1] -0.33072101  1.22179038 -0.24173769 -1.26257410 -0.04540784
##
## [[3]]
## [1] -0.6339931  0.3043004  0.5044343 -0.6714019 -1.0028538
##
## [[4]]
## [1] -0.05054396  1.29137136  1.59884373  0.70757312 -0.54994101
##
## [[5]]
## [1] -0.2437190 -0.4938076  0.5475734  0.2745613  0.6143301
##
## [[6]]
## [1] -0.30126009  0.07438473 -0.42646775 -0.47477253 -0.50431541
##
## [[7]]
## [1]  0.004690147  0.227253698 -0.166094132  1.292331900 -0.689665017
##
## [[8]]
## [1] -0.2253978  0.2012822 -0.2727642 -0.6524616 -0.4527512
```

```
set.seed(42)
mclapply(1:ncores, function(x) rnorm(5), mc.cores = ncores)
```

```
## [[1]]
```

```
## [1]  1.443889  1.272290 -2.091843 -1.378653  1.123251
##
## [[2]]
## [1] -0.3118257  0.7424002 -1.8022330 -1.0509128  2.3689426
##
## [[3]]
## [1] -0.02501858 -0.24392659  1.09840222 -1.95574182  0.83341149
##
## [[4]]
## [1] -0.6013366  2.2874768  1.3982151  0.6849199 -0.3456062
##
## [[5]]
## [1]  0.3064059  0.3798483  1.8715021 -0.8857134  0.6057915
##
## [[6]]
## [1] -1.15339549 -1.32182246  0.94920562 -0.02896595 -0.80759039
##
## [[7]]
## [1] -1.35368488  0.04857132 -2.25851991  1.77966712 -0.25228998
##
## [[8]]
## [1] -0.8240983 -0.2797501 -0.1128762 -0.3914366  1.1527707
```

We don't have reproducibility.

8.1.1.2 Try 2

```
set.seed(42)
mclapply(1:ncores, function(x) rnorm(5), mc.cores = ncores, mc.set.seed = FALSE)
```

```
## [[1]]
## [1]  1.3709584 -0.5646982  0.3631284  0.6328626  0.4042683
##
## [[2]]
## [1]  1.3709584 -0.5646982  0.3631284  0.6328626  0.4042683
##
## [[3]]
## [1]  1.3709584 -0.5646982  0.3631284  0.6328626  0.4042683
##
## [[4]]
## [1]  1.3709584 -0.5646982  0.3631284  0.6328626  0.4042683
##
## [[5]]
## [1]  1.3709584 -0.5646982  0.3631284  0.6328626  0.4042683
##
## [[6]]
## [1]  1.3709584 -0.5646982  0.3631284  0.6328626  0.4042683
##
## [[7]]
## [1]  1.3709584 -0.5646982  0.3631284  0.6328626  0.4042683
##
## [[8]]
## [1]  1.3709584 -0.5646982  0.3631284  0.6328626  0.4042683
```

```
set.seed(42)
mclapply(1:ncores, function(x) rnorm(5), mc.cores = ncores, mc.set.seed = FALSE)
```

```
## [[1]]
## [1]  1.3709584 -0.5646982  0.3631284  0.6328626  0.4042683
##
## [[2]]
## [1]  1.3709584 -0.5646982  0.3631284  0.6328626  0.4042683
##
## [[3]]
## [1]  1.3709584 -0.5646982  0.3631284  0.6328626  0.4042683
##
## [[4]]
## [1]  1.3709584 -0.5646982  0.3631284  0.6328626  0.4042683
##
## [[5]]
## [1]  1.3709584 -0.5646982  0.3631284  0.6328626  0.4042683
##
## [[6]]
## [1]  1.3709584 -0.5646982  0.3631284  0.6328626  0.4042683
##
## [[7]]
## [1]  1.3709584 -0.5646982  0.3631284  0.6328626  0.4042683
##
## [[8]]
## [1]  1.3709584 -0.5646982  0.3631284  0.6328626  0.4042683
```

We have reproducibility, but we don't have different random number streams for the different processes.

8.1.1.3 Try 3

```
RNGkind("L'Ecuyer-CMRG")
set.seed(42)
mclapply(1:ncores, function(x) rnorm(5), mc.cores = ncores)
```

```
## [[1]]
## [1]  1.11932846 -0.07617141 -0.35021912 -0.33491161 -1.73311280
##
## [[2]]
## [1] -0.2084809 -1.0341493 -0.2629060  0.3880115  0.8331067
##
## [[3]]
## [1]  0.001100034  1.763058291 -0.166377859 -0.311947389  0.694879494
##
## [[4]]
## [1]  0.2262605 -0.4827515  1.7637105 -0.1887217 -0.7998982
##
## [[5]]
## [1]  0.8584220 -0.3851236  1.0817530  0.2851169  0.1799325
##
## [[6]]
## [1] -1.1378621 -1.5197576 -0.9198612  1.0303683 -0.9458347
##
## [[7]]
```

```
## [1] -0.04649149  3.38053730 -0.35705061  0.17722940 -0.39716405
##
## [[8]]
## [1]  1.3502819 -1.0055894 -0.4591798 -0.0628527 -0.2706805
```

```
set.seed(42)
mclapply(1:ncores, function(x) rnorm(5), mc.cores = ncores)
```

```
## [[1]]
## [1]  1.11932846 -0.07617141 -0.35021912 -0.33491161 -1.73311280
##
## [[2]]
## [1] -0.2084809 -1.0341493 -0.2629060  0.3880115  0.8331067
##
## [[3]]
## [1]  0.001100034  1.763058291 -0.166377859 -0.311947389  0.694879494
##
## [[4]]
## [1]  0.2262605 -0.4827515  1.7637105 -0.1887217 -0.7998982
##
## [[5]]
## [1]  0.8584220 -0.3851236  1.0817530  0.2851169  0.1799325
##
## [[6]]
## [1] -1.1378621 -1.5197576 -0.9198612  1.0303683 -0.9458347
##
## [[7]]
## [1] -0.04649149  3.38053730 -0.35705061  0.17722940 -0.39716405
##
## [[8]]
## [1]  1.3502819 -1.0055894 -0.4591798 -0.0628527 -0.2706805
```

Just right! We have different random numbers in all our jobs. And it is reproducible.

8.1.1.4 Try 4

But this does not work like you may think it does.

```
save.seed <- .Random.seed
mclapply(1:ncores, function(x) rnorm(5), mc.cores = ncores)
```

```
## [[1]]
## [1]  1.11932846 -0.07617141 -0.35021912 -0.33491161 -1.73311280
##
## [[2]]
## [1] -0.2084809 -1.0341493 -0.2629060  0.3880115  0.8331067
##
## [[3]]
## [1]  0.001100034  1.763058291 -0.166377859 -0.311947389  0.694879494
##
## [[4]]
## [1]  0.2262605 -0.4827515  1.7637105 -0.1887217 -0.7998982
##
## [[5]]
## [1]  0.8584220 -0.3851236  1.0817530  0.2851169  0.1799325
##
```

```
## [[6]]
## [1] -1.1378621 -1.5197576 -0.9198612  1.0303683 -0.9458347
##
## [[7]]
## [1] -0.04649149  3.38053730 -0.35705061  0.17722940 -0.39716405
##
## [[8]]
## [1]  1.3502819 -1.0055894 -0.4591798 -0.0628527 -0.2706805
```

```
identical(save.seed, .Random.seed)
```

```
## [1] TRUE
```

Running `mclapply` does not change `.Random.seed` in the parent process (the R process you are typing into). It only changes it in the child processes (that do the work). But there is no communication from child to parent *except* the list of results returned by `mclapply`.

This is a fundamental problem with `mclapply` and the fork-exec method of parallelization. And it has no real solution. The different child processes are using different random number streams (we see that, and it is what we wanted to happen). So they should all have a different `.Random.seed` at the end. Let's check.

```
fred <- function(x) {
    sally <- rnorm(5)
    list(normals = sally, seeds = .Random.seed)
}
mclapply(1:ncores, fred, mc.cores = ncores)
```

```
## [[1]]
## [[1]]$normals
## [1]  1.11932846 -0.07617141 -0.35021912 -0.33491161 -1.73311280
##
## [[1]]$seeds
## [1]         407   843426105 -1189635545 -1908310047  -500321376 -1368039072
## [7]  -327247439
##
##
## [[2]]
## [[2]]$normals
## [1] -0.2084809 -1.0341493 -0.2629060  0.3880115  0.8331067
##
## [[2]]$seeds
## [1]        407 -846419547  937862459 1326107580  760134144 1807130611 1335532618
##
##
## [[3]]
## [[3]]$normals
## [1]  0.001100034  1.763058291 -0.166377859 -0.311947389  0.694879494
##
## [[3]]$seeds
## [1]         407  -133207412  1779373486   976163781  -458962600 -1469488387
## [7] -2147451017
##
##
## [[4]]
## [[4]]$normals
## [1]  0.2262605 -0.4827515  1.7637105 -0.1887217 -0.7998982
```

```
##
## [[4]]$seeds
## [1]         407 -1395279340 -1740117305 -2068775159 -1223675294  1644811352
## [7]   970811783
##
##
## [[5]]
## [[5]]$normals
## [1]  0.8584220 -0.3851236  1.0817530  0.2851169  0.1799325
##
## [[5]]$seeds
## [1]         407 -1988850249   851836302  -412123035 -1393827477 -1602296088
## [7] -1726579968
##
##
## [[6]]
## [[6]]$normals
## [1] -1.1378621 -1.5197576 -0.9198612  1.0303683 -0.9458347
##
## [[6]]$seeds
## [1]         407  -289872396 -1983423440 -1372057278  1254597746  1572309401
## [7]  -139829874
##
##
## [[7]]
## [[7]]$normals
## [1] -0.04649149  3.38053730 -0.35705061  0.17722940 -0.39716405
##
## [[7]]$seeds
## [1]         407  -552026993  1357891868 -1020113790  1248945061  -126548789
## [7]  -322082143
##
##
## [[8]]
## [[8]]$normals
## [1]  1.3502819 -1.0055894 -0.4591798 -0.0628527 -0.2706805
##
## [[8]]$seeds
## [1]         407 -1867726870  1386420444  -153979515 -1662972776  -302869263
## [7]  -198526739
```

Right! Conceptually, there is no Right Thing to do! We want to advance the RNG seed in the parent process, but to what? We have four different possibilities (with four child processes), but we only want one answer, not four!

So the only solution to this problem is not really a solution. You just have to be aware of the issue. If you want to do exactly the same random thing with `mclapply` and get different random results, then you must change `.Random.seed` in the parent process, either with `set.seed` or by otherwise using random numbers *in the parent process*.

8.1.2 The Example

We need to rewrite our `doit` function

- to only do `1 / ncores` of the work in each child process,

- to not set the random number generator seed, and
- to take an argument in some list we provide.

```
doit <- function(nsim, estimator) {
    result <- double(nsim)
    for (i in 1:nsim) {
        x <- rnorm(n, theta, abs(theta))
        result[i] <- estimator(x)
    }
    return(result)
}
```

8.1.3 Try It

```
mout <- mclapply(rep(nsim / ncores, ncores), doit,
    estimator = mle, mc.cores = ncores)
lapply(mout, head)
```

```
## [[1]]
## [1] 0.9051972 0.9589889 0.9799828 1.1347548 0.9090886 0.9821320
##
## [[2]]
## [1] 0.8317815 1.3432331 0.7821308 1.2010078 0.9792244 1.1148521
##
## [[3]]
## [1] 0.8627829 0.9790400 1.1787975 0.7852431 1.2942963 1.0768396
##
## [[4]]
## [1] 1.0422013 0.9166641 0.8326720 1.1864809 0.9609456 1.3137716
```

8.1.4 Check It

Seems to have worked.

```
length(mout)
```

```
## [1] 4
```

```
sapply(mout, length)
```

```
## [1] 25000 25000 25000 25000
```

```
lapply(mout, head)
```

```
## [[1]]
## [1] 0.9051972 0.9589889 0.9799828 1.1347548 0.9090886 0.9821320
##
## [[2]]
## [1] 0.8317815 1.3432331 0.7821308 1.2010078 0.9792244 1.1148521
##
## [[3]]
## [1] 0.8627829 0.9790400 1.1787975 0.7852431 1.2942963 1.0768396
##
## [[4]]
```

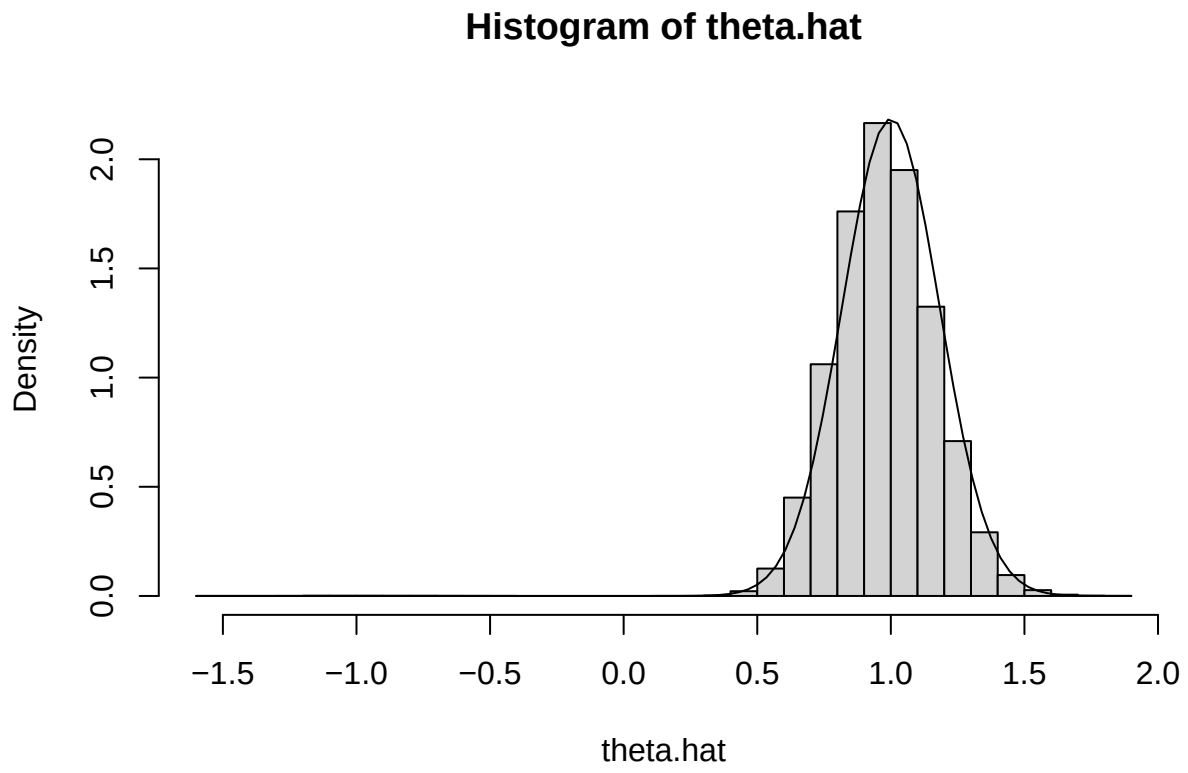
```
## [1] 1.0422013 0.9166641 0.8326720 1.1864809 0.9609456 1.3137716
```

```
lapply(mout, range)
```

```
## [[1]]
## [1] -1.598078  1.742441
##
## [[2]]
## [1] -1.150278  1.878888
##
## [[3]]
## [1] -1.284372  1.834568
##
## [[4]]
## [1] -1.366671  1.747764
```

Plot it.

```
theta.hat <- unlist(mout)
hist(theta.hat, probability = TRUE, breaks = 30)
curve(dnorm(x, mean = theta, sd = theta / sqrt(3 * n)), add = TRUE)
```



8.1.5 Time It

```
time4 <- NULL
for (irep in 1:nrep)
    time4 <- rbind(time4, system.time(theta.hat.mle <-
        unlist(mclapply(rep(nsim / ncores, ncores), doit,
            estimator = mle, mc.cores = ncores))))
time4
```

```
##      user.self sys.self elapsed user.child sys.child
## [1,]     0.004    0.008   3.913     11.276     0.060
## [2,]     0.004    0.008   3.792     14.840     0.056
## [3,]     0.000    0.008   3.767     14.792     0.080
## [4,]     0.000    0.012   3.738     14.760     0.100
## [5,]     0.000    0.008   3.748     14.780     0.052
## [6,]     0.004    0.004   3.743     14.768     0.092
## [7,]     0.000    0.012   3.876     14.788     0.096
```

```
apply(time4, 2, mean)
```

```
##    user.self     sys.self      elapsed   user.child    sys.child
##  0.001714286  0.008571429  3.796714286 14.286285714  0.076571429
```

```
apply(time4, 2, sd) / sqrt(nrep)
```

```
##   user.self    sys.self     elapsed  user.child   sys.child
## 0.000808122 0.001043281 0.026466254 0.501808430 0.007680703
```

We got the desired speedup. The elapsed time averages

```
apply(time4, 2, mean)["elapsed"]
```

```
##  elapsed
## 3.796714
```

with parallelization and

```
apply(time2, 2, mean)["elapsed"]
```

```
##  elapsed
## 7.728286
```

without parallelization. But we did not get a 4-fold speedup with 4 cores. There is a cost to starting and stopping the child processes. And some time needs to be taken from this number crunching to run the rest of the computer. However, we did get an almost 3-fold speedup. If we had more cores, we could do even better.

8.2 The Example With a Cluster

This method is more complicated but

- it works on clusters like the ones at the Minnesota Supercomputing Institute or at LATIS (College of Liberal Arts Technologies and Innovation Services, and
- according to the documentation, it does work on Windows.

First a toy problem that does nothing except show that we are actually using different processes.

```
library(parallel)
ncores <- detectCores()
cl <- makePSOCKcluster(ncores)
parLapply(cl, 1:ncores, function(x) Sys.getpid())
```

```
## [[1]]
## [1] 13573
##
## [[2]]
## [1] 13576
##
## [[3]]
```

```
## [1] 13577
##
## [[4]]
## [1] 13574
##
## [[5]]
## [1] 13575
##
## [[6]]
## [1] 13572
##
## [[7]]
## [1] 13571
##
## [[8]]
## [1] 13578
```

```
stopCluster(cl)
```

This is more complicated in that

- first you you set up a cluster, here with `makePSOCKcluster` but not everywhere — there are a variety of different commands to make clusters and the command would be different at MSI or LATIS — and
- at the end you tear down the cluster with `stopCluster`.

Of course, you do not need to tear down the cluster before you are done with it. You can execute multiple `parLapply` commands on the same cluster.

There are also a lot of other commands other than `parLapply` that can be used on the cluster. We will see some of them below.

8.2.1 Parallel Streams of Random Numbers

```
cl <- makePSOCKcluster(ncores)
clusterSetRNGStream(cl, 42)
parLapply(cl, 1:ncores, function(x) rnorm(5))
```

```
## [[1]]
## [1] -0.93907708 -0.04167943  0.82941349 -0.43935820 -0.31403543
##
## [[2]]
## [1]  1.11932846 -0.07617141 -0.35021912 -0.33491161 -1.73311280
##
## [[3]]
## [1] -0.2084809 -1.0341493 -0.2629060  0.3880115  0.8331067
##
## [[4]]
## [1]  0.001100034  1.763058291 -0.166377859 -0.311947389  0.694879494
##
## [[5]]
## [1]  0.2262605 -0.4827515  1.7637105 -0.1887217 -0.7998982
##
## [[6]]
## [1]  0.8584220 -0.3851236  1.0817530  0.2851169  0.1799325
##
```

```
## [[7]]
## [1] -1.1378621 -1.5197576 -0.9198612  1.0303683 -0.9458347
##
## [[8]]
## [1] -0.04649149  3.38053730 -0.35705061  0.17722940 -0.39716405
```

```
parLapply(cl, 1:ncores, function(x) rnorm(5))
```

```
## [[1]]
## [1] -2.1290236  2.5069224 -1.1273128  0.1660827  0.5767232
##
## [[2]]
## [1] 0.03628534 0.29647473 1.07128138 0.72844380 0.12458507
##
## [[3]]
## [1] -0.1652167 -0.3262253 -0.2657667  0.1878883  1.4916193
##
## [[4]]
## [1]  0.3541931 -0.6820627 -1.0762411 -0.9595483  0.0982342
##
## [[5]]
## [1]  0.5441483  1.0852866  1.6011037 -0.5018903 -0.2709106
##
## [[6]]
## [1] -0.57445721 -0.86440961 -0.77401840  0.54423137 -0.01006838
##
## [[7]]
## [1] -1.3057289  0.5911102  0.8416164  1.7477622 -0.7824792
##
## [[8]]
## [1]  0.9071634  0.2518615 -0.4905999  0.4900700  0.7970189
```

We see that clusters do not have the same problem with continuing random number streams that the fork-exec mechanism has.

- Using fork-exec there is a *parent* process and *child* processes (all running on the same computer) and the *child* processes end when their work is done (when `mclapply` finishes).
- Using clusters there is a *controller* process and *worker* processes (possibly running on many different computers) and the *worker* processes end when the cluster is torn down (with `stopCluster`).

So the worker processes continue and remember where they are in the random number stream.

8.2.2 The Example on a Cluster

8.2.2.1 Set Up

Another complication of using clusters is that the worker processes are completely independent of the controller process. Any information they have must be explicitly passed to them.

This is very unlike the fork-exec model in which all of the child processes are copies of the parent process inheriting all of its memory (and thus knowing about any and all R objects it created).

So in order for our example to work we must explicitly distribute stuff to the cluster.

```
clusterExport(cl, c("doit", "mle", "mlogl", "n", "nsim", "theta"))
```

Now all of the workers have those R objects, as copied from the controller process right now. If we change them in the controller (pedantically if we change the R objects those *names* refer to) the workers won't know about it. They only would get access to those changes if code were executed on them to do so.

8.2.2.2 Try It

So now we are set up to try our example.

```
pout <- parLapply(cl, rep(nsim / ncores, ncores), doit, estimator = mle)
```

8.2.2.3 Check It

Seems to have worked.

```
length(pout)
```

```
## [1] 4
```

```
sapply(pout, length)
```

```
## [1] 25000 25000 25000 25000
```

```
lapply(pout, head)
```

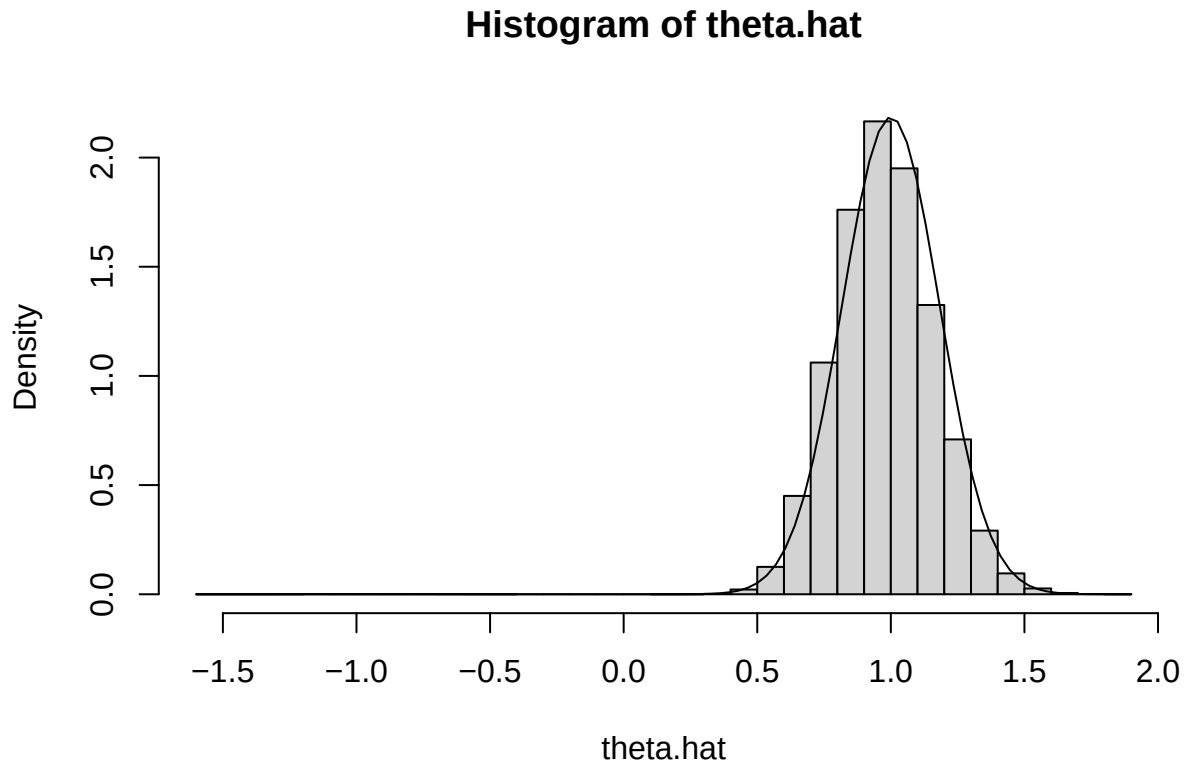
```
## [[1]]
## [1] 1.0079313 0.7316543 0.4958322 0.7705943 0.7734226 0.6158992
##
## [[2]]
## [1] 0.9589889 0.9799828 1.1347548 0.9090886 0.9821320 1.0032531
##
## [[3]]
## [1] 1.3432331 0.7821308 1.2010078 0.9792244 1.1148521 0.9269000
##
## [[4]]
## [1] 0.9790400 1.1787975 0.7852431 1.2942963 1.0768396 0.7546295
```

```
lapply(pout, range)
```

```
## [[1]]
## [1] -1.379124  1.800773
##
## [[2]]
## [1] -1.598078  1.742441
##
## [[3]]
## [1] -1.150278  1.878888
##
## [[4]]
## [1] -1.284372  1.834568
```

Plot it.

```
theta.hat <- unlist(mout)
hist(theta.hat, probability = TRUE, breaks = 30)
curve(dnorm(x, mean = theta, sd = theta / sqrt(3 * n)), add = TRUE)
```



8.2.2.4 Time It

```
time5 <- NULL
for (irep in 1:nrep)
    time5 <- rbind(time5, system.time(theta.hat.mle <-
        unlist(parLapply(cl, rep(nsim / ncores, ncores),
            doit, estimator = mle))))
time5
```

```
##      user.self sys.self elapsed user.child sys.child
## [1,]     0.004    0.000   4.032          0         0
## [2,]     0.008    0.000   6.946          0         0
## [3,]     0.000    0.004   4.235          0         0
## [4,]     0.004    0.000   4.151          0         0
## [5,]     0.000    0.000   4.172          0         0
## [6,]     0.004    0.000   4.073          0         0
## [7,]     0.004    0.000   4.007          0         0
```

```
apply(time5, 2, mean)
```

```
##    user.self     sys.self      elapsed   user.child    sys.child
## 0.0034285714 0.0005714286 4.5165714286 0.0000000000 0.0000000000
```

```
apply(time5, 2, sd) / sqrt(nrep)
```

```
##    user.self     sys.self      elapsed   user.child    sys.child
## 0.0010432811 0.0005714286 0.4060555238 0.0000000000 0.0000000000
```

We got the desired speedup. The elapsed time averages

```
apply(time5, 2, mean)["elapsed"]
```

```
##  elapsed
## 4.516571
```

with parallelization and

```
apply(time2, 2, mean)["elapsed"]
```

```
##  elapsed
## 7.728286
```

without parallelization. But we did not get a 4-fold speedup with 4 cores. There is a cost to sending information to and from the worker processes. And some time needs to be taken from this number crunching to run the rest of the computer. However, we did get a 2-fold speedup. If we had more workers, we could do even better.

8.2.3 Tear Down

Don't forget to tear down the cluster when you are done.

```
stopCluster(cl)
```