Stat 3701 Lecture Notes: Parallel Computing in R

Charles J. Geyer

December 02, 2022

1 License

2 Note

These notes are the course material for the undergraduate statistical computing course Stat 3701. The best version of my class notes for parallel computing are those for Stat 8054 (PhD level statistical computing). They are, however, terser.

3 R

- The version of R used to make this document is 4.2.1.
- The version of the `rmarkdown` package used to make this document is 2.17.

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 16093280 kB (8 GB, which is 250,000 times as much as the IBM 1130). Its clock speed is 2.40GHz (8,633 times as fast).

That is `log2(8633)` = 13.08 doublings of speed in 51 years, which is a doubling of computer speed every 3.9 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 8 separate processes (two hyperthreads per core) at (nearly) full machine speed. The desktop in my office has an Intel i7 which is similar (8 hyperthreads) but faster. 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

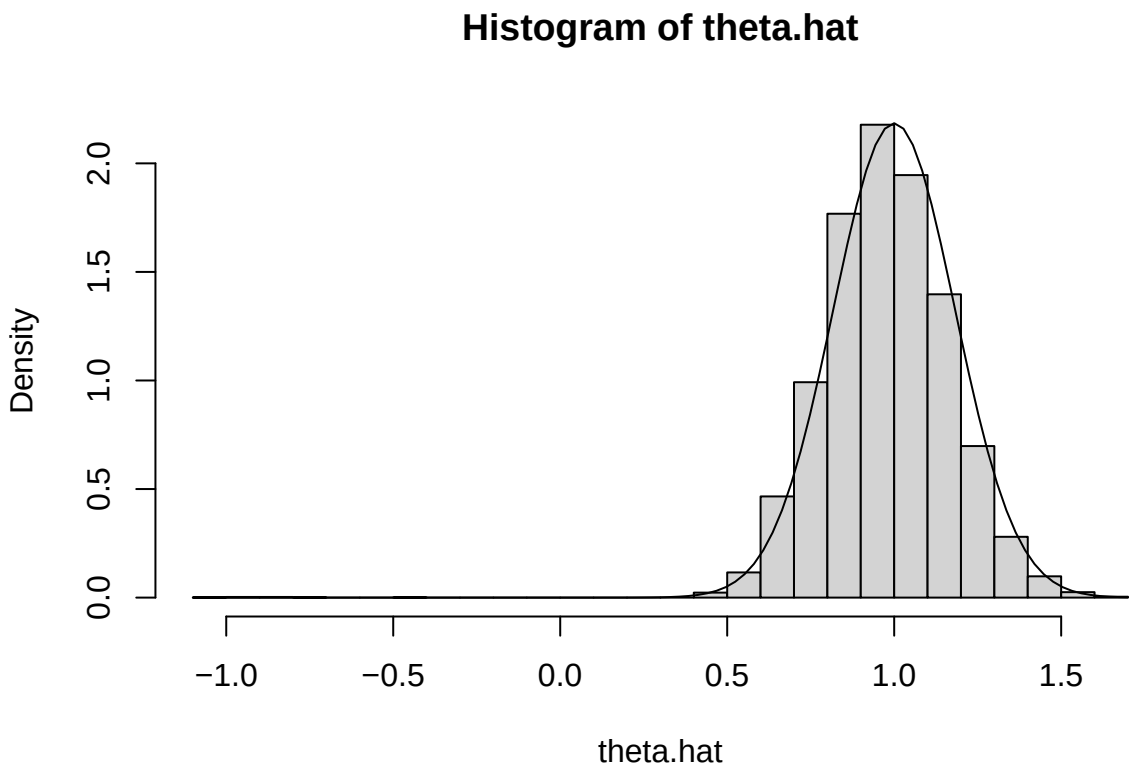
```
n <- 10
nsim <- 1e4
theta <- 1

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
##   0.809   0.000   0.809
```

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,]     8.345    0.000   8.344          0         0
## [2,]     8.411    0.004   8.414          0         0
## [3,]     8.344    0.000   8.344          0         0
## [4,]     8.164    0.000   8.163          0         0
## [5,]     8.076    0.004   8.080          0         0
## [6,]     8.284    0.000   8.284          0         0
## [7,]     8.052    0.000   8.051          0         0
```

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

```
##   user.self    sys.self     elapsed  user.child   sys.child
## 8.239428571 0.001142857 8.240000000 0.000000000 0.000000000
```

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

```
##    user.self     sys.self      elapsed   user.child    sys.child
## 0.0536815376 0.0007377111 0.0536953487 0.0000000000 0.0000000000
```

7 Parallel Computing

7.1 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 unless you use it with no parallelization, optional argument `mc.cores = 1` or unless you are running R under Windows Subsystem for Linux (WSL), which is a complete implementation of Linux running inside Microsoft Windows.

  If you want to do this example on Windows without WSL, use `mc.cores = 1`.

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] 317838
##
## [[2]]
## [1] 317839
```

```
##
## [[3]]
## [1] 317840
##
## [[4]]
## [1] 317841
##
## [[5]]
## [1] 317842
##
## [[6]]
## [1] 317843
##
## [[7]]
## [1] 317844
##
## [[8]]
## [1] 317845
```

7.1.1 Parallel Streams of Random Numbers

7.1.1.1 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]  0.09101567  0.39496221 -0.17652490 -1.01115455 -0.05414983
##
## [[2]]
## [1]  0.6908300  0.3255392 -0.4195909 -1.6149470 -0.0543658
##
## [[3]]
## [1] -0.43117106 -0.27364046  0.03727203 -0.08873792  0.09104350
##
## [[4]]
## [1] -0.57874095 -1.08547662 -0.03040493  0.44642133 -1.58528836
##
## [[5]]
## [1] -1.80852984 -0.09869252 -0.85926312 -0.07319315  0.38319201
##
## [[6]]
## [1] -1.2353376 -3.2161033 -0.9529549  1.4211435  0.4874526
##
## [[7]]
## [1] 0.1227484 2.1710417 0.7712976 0.9807209 0.8199643
##
## [[8]]
## [1]  0.5095189 -0.5873218  2.4725626  0.1608429 -1.2524278
```

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

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

```
## [1] -0.4853520 -1.0645395 -1.9353793  0.8412434  2.1423030
##
## [[2]]
## [1]  1.3201150  0.5121410  0.4841618 -1.1009975 -0.8221320
##
## [[3]]
## [1]  1.1300011 -1.3224798 -1.1066912 -0.6879495 -0.3958112
##
## [[4]]
## [1]  0.1834762 -0.6508783 -1.1375858 -0.3594214 -0.1399525
##
## [[5]]
## [1]  0.8348606 -0.8213994  1.7235827 -0.9028726  0.1606350
##
## [[6]]
## [1] -0.7482060  0.5680813 -0.8968114 -0.1352857  0.7232129
##
## [[7]]
## [1]  1.1361550 -0.4808321  0.4343621  0.2006515  1.2696169
##
## [[8]]
## [1] -0.2718976  0.9534515 -0.9828857 -1.0600107 -0.5587632
```

We don't have reproducibility.

7.1.1.2 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.

7.1.1.3 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.

7.1.1.4 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]       10407   843426105 -1189635545 -1908310047  -500321376 -1368039072
## [7]  -327247439
##
##
## [[2]]
## [[2]]$normals
## [1] -0.2084809 -1.0341493 -0.2629060  0.3880115  0.8331067
##
## [[2]]$seeds
## [1]      10407 -846419547  937862459 1326107580  760134144 1807130611 1335532618
##
##
## [[3]]
## [[3]]$normals
## [1]  0.001100034  1.763058291 -0.166377859 -0.311947389  0.694879494
##
## [[3]]$seeds
## [1]       10407  -133207412  1779373486   976163781  -458962600 -1469488387
## [7] -2147451017
##
##
## [[4]]
## [[4]]$normals
## [1]  0.2262605 -0.4827515  1.7637105 -0.1887217 -0.7998982
##
```

```
## [[4]]$seeds
## [1]       10407 -1395279340 -1740117305 -2068775159 -1223675294  1644811352
## [7]   970811783
##
##
## [[5]]
## [[5]]$normals
## [1]  0.8584220 -0.3851236  1.0817530  0.2851169  0.1799325
##
## [[5]]$seeds
## [1]       10407 -1988850249   851836302  -412123035 -1393827477 -1602296088
## [7] -1726579968
##
##
## [[6]]
## [[6]]$normals
## [1] -1.1378621 -1.5197576 -0.9198612  1.0303683 -0.9458347
##
## [[6]]$seeds
## [1]       10407  -289872396 -1983423440 -1372057278  1254597746  1572309401
## [7]  -139829874
##
##
## [[7]]
## [[7]]$normals
## [1] -0.04649149  3.38053730 -0.35705061  0.17722940 -0.39716405
##
## [[7]]$seeds
## [1]       10407  -552026993  1357891868 -1020113790  1248945061  -126548789
## [7]  -322082143
##
##
## [[8]]
## [[8]]$normals
## [1]  1.3502819 -1.0055894 -0.4591798 -0.0628527 -0.2706805
##
## [[8]]$seeds
## [1]       10407 -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 eight different possibilities (with eight child processes), but we only want one answer, not eight!

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.*

7.1.2 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)
}
```

7.1.3 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
##
## [[5]]
## [1] 0.8057316 0.9488173 1.0792078 0.9774531 0.8106612 0.8403027
##
## [[6]]
## [1] 1.0156983 1.0077599 0.9867766 1.1643493 0.9478923 1.1770221
##
## [[7]]
## [1] 1.2287013 1.0046353 0.9560784 1.0354414 0.9045423 0.9455714
##
## [[8]]
## [1] 0.7768910 1.0376265 0.8830854 0.8911714 1.0288567 1.1609360
```

7.1.4 Check It

Seems to have worked.

```
length(mout)
```

```
## [1] 8
```

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

```
## [1] 12500 12500 12500 12500 12500 12500 12500 12500
```

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

```
## [[1]]
## [1] 0.9051972 0.9589889 0.9799828 1.1347548 0.9090886 0.9821320
```

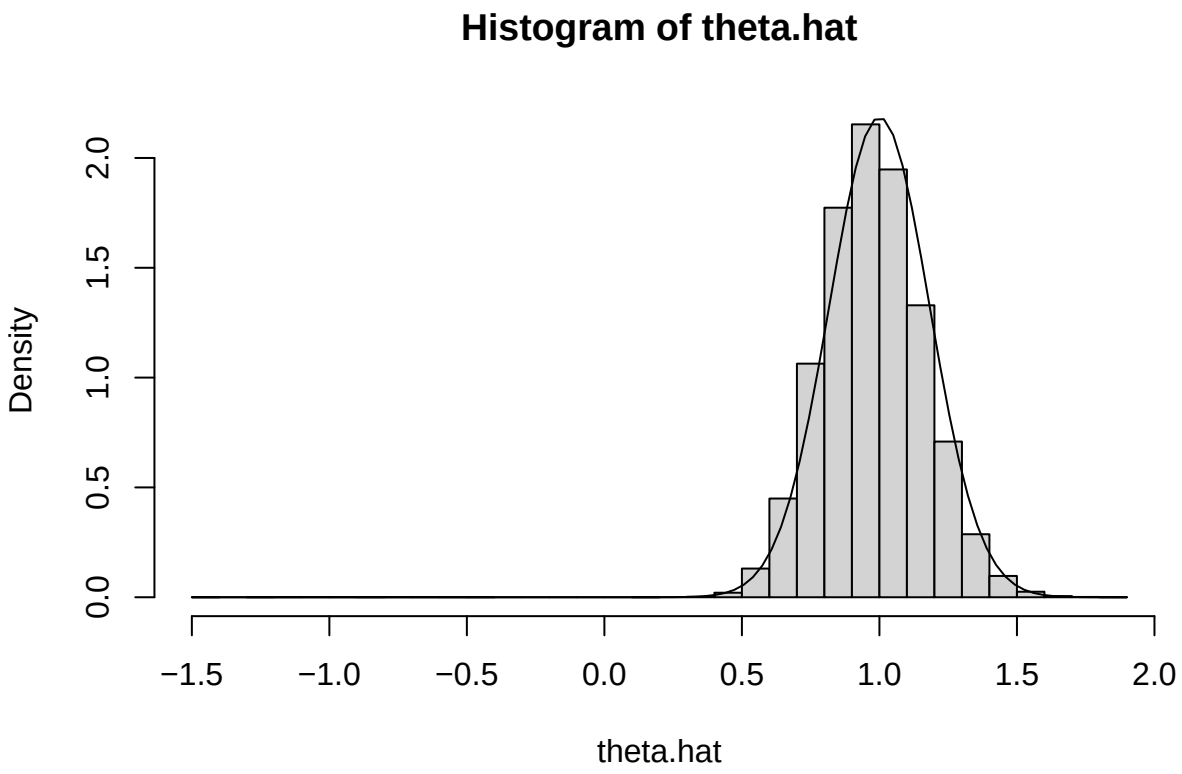
```
##
## [[2]]
## [1] 0.8317815 1.3432331 0.7821308 1.2010078 0.9792244 1.1148521
##
## [[3]]
## [1] 0.8627829 0.9790400 1.1787975 0.7852431 1.2942963 1.0768396
##
## [[4]]
## [1] 1.0422013 0.9166641 0.8326720 1.1864809 0.9609456 1.3137716
##
## [[5]]
## [1] 0.8057316 0.9488173 1.0792078 0.9774531 0.8106612 0.8403027
##
## [[6]]
## [1] 1.0156983 1.0077599 0.9867766 1.1643493 0.9478923 1.1770221
##
## [[7]]
## [1] 1.2287013 1.0046353 0.9560784 1.0354414 0.9045423 0.9455714
##
## [[8]]
## [1] 0.7768910 1.0376265 0.8830854 0.8911714 1.0288567 1.1609360
```

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

```
## [[1]]
## [1] -1.452060  1.742441
##
## [[2]]
## [1] -1.104573  1.878888
##
## [[3]]
## [1] -1.164338  1.800782
##
## [[4]]
## [1] -1.195419  1.707977
##
## [[5]]
## [1] -1.275307  1.785461
##
## [[6]]
## [1] -1.115946  1.773345
##
## [[7]]
## [1] -1.045440  1.774671
##
## [[8]]
## [1] -1.405485  1.662768
```

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)
```



7.1.5 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,]     0.006    0.021   1.966     12.609     0.276
## [2,]     0.002    0.025   1.988     12.830     0.232
## [3,]     0.002    0.023   1.992     10.806     0.235
## [4,]     0.002    0.024   1.975     12.621     0.291
## [5,]     0.002    0.024   2.009     12.749     0.293
## [6,]     0.003    0.025   2.038     12.979     0.247
## [7,]     0.002    0.024   2.008     10.984     0.249
```

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

```
##    user.self     sys.self      elapsed   user.child    sys.child
##  0.002714286  0.023714286  1.996571429 12.225428571  0.260428571
```

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

```
##    user.self     sys.self      elapsed   user.child    sys.child
## 0.0005654449 0.0005216405 0.0091231186 0.3473378249 0.0097635306
```

We got the desired speedup. The elapsed time averages

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

```
##  elapsed
## 1.996571
```

with parallelization and

```
apply(time1, 2, mean)["elapsed"]
```

```
## elapsed
##    8.24
```

without parallelization. But we did not get a 8-fold speedup with 8 hyperthreads. 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 slightly more than a 4-fold speedup. If we had more cores in our machine, we could do even better.

7.2 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] 318004
##
## [[2]]
## [1] 318002
##
## [[3]]
## [1] 318001
##
## [[4]]
## [1] 318005
##
## [[5]]
## [1] 318007
##
## [[6]]
## [1] 318003
##
## [[7]]
## [1] 318008
##
## [[8]]
## [1] 318006
```

```
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.

7.2.1 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.

7.2.2 The Example on a Cluster

7.2.2.1 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.

7.2.2.2 Try It

So now we are set up to try our example.

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

7.2.2.3 Check It

Seems to have worked.

```
length(pout)
```

```
## [1] 8
```

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

```
## [1] 12500 12500 12500 12500 12500 12500 12500 12500
```

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

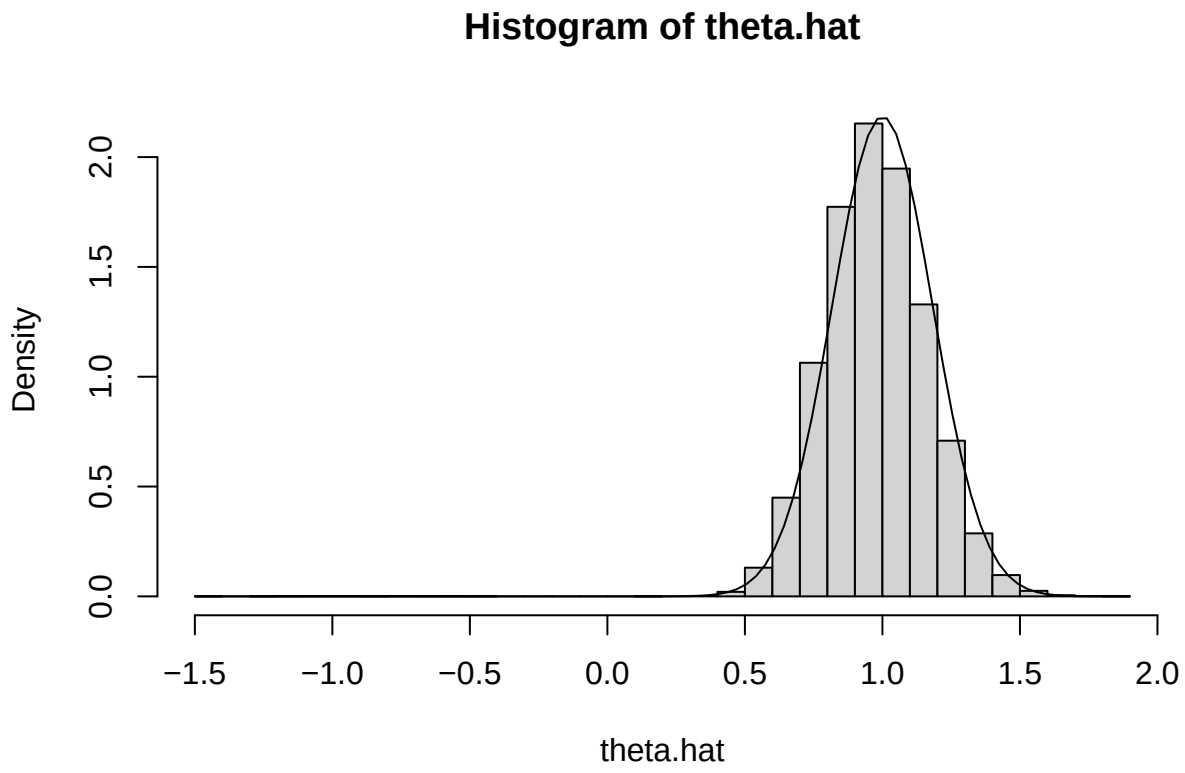
```
## [[1]]
## [1] 1.0079313 0.7316543 0.4958322 0.7705943 0.7734226 0.6158992
##
## [[2]]
## [1] 0.9589889 0.9799828 1.1347548 0.9090886 0.9821320 1.0032531
##
## [[3]]
## [1] 1.3432331 0.7821308 1.2010078 0.9792244 1.1148521 0.9269000
##
## [[4]]
## [1] 0.9790400 1.1787975 0.7852431 1.2942963 1.0768396 0.7546295
##
## [[5]]
## [1] 0.9166641 0.8326720 1.1864809 0.9609456 1.3137716 0.9832663
##
## [[6]]
## [1] 0.9488173 1.0792078 0.9774531 0.8106612 0.8403027 1.1296857
##
## [[7]]
## [1] 1.0077599 0.9867766 1.1643493 0.9478923 1.1770221 1.2789464
##
## [[8]]
## [1] 1.0046353 0.9560784 1.0354414 0.9045423 0.9455714 1.0312553
```

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

```
## [[1]]
## [1] -1.339373  1.800773
##
## [[2]]
## [1] -1.452060  1.742441
##
## [[3]]
## [1] -1.104573  1.878888
##
## [[4]]
## [1] -1.164338  1.800782
##
## [[5]]
## [1] -1.195419  1.707977
##
## [[6]]
## [1] -1.275307  1.785461
##
## [[7]]
## [1] -1.115946  1.773345
##
## [[8]]
## [1] -1.045440  1.774671
```

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)
```



7.2.2.4 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,]     0.001    0.004   2.030          0         0
## [2,]     0.005    0.000   2.005          0         0
## [3,]     0.002    0.004   2.033          0         0
## [4,]     0.004    0.000   2.039          0         0
## [5,]     0.006    0.000   2.002          0         0
## [6,]     0.005    0.000   2.047          0         0
## [7,]     0.007    0.001   2.047          0         0
```

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

```
##   user.self    sys.self     elapsed  user.child   sys.child
## 0.004285714 0.001285714 2.029000000 0.000000000 0.000000000
```

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

```
##    user.self     sys.self      elapsed   user.child    sys.child
## 0.0008081220 0.0007142857 0.0070203785 0.0000000000 0.0000000000
```

We got the desired speedup. The elapsed time averages

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

```
## elapsed
##   2.029
```

with parallelization and

```
apply(time1, 2, mean)["elapsed"]
```

```
## elapsed
##    8.24
```

without parallelization. But we did not get a 8-fold speedup with 8 cores (actually hyperthreads). 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 slightly more than a 4-fold speedup. If we had more workers that could run simultaneously (like on a cluster at LATIS or at the supercomputer center), we could do even better.

7.2.3 Tear Down

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

```
stopCluster(cl)
```