Introduction to parallel processing

Timothée Bacri
University of Bergen

06 December, 2022

Section 1

How to achieve high performance

Specialization

Hardware

- Powerful CPU
- Recent CPU
- CPUs with instruction sets AES-NI / AVX / other
- GPUs for image processing

Software

- `ADMB`, `TMB`
- BLAS (Basic Linear Algebra Subroutines) libraries for basic vector and matrix operations
- LAPACK (Linear Algebra Package) libraries for solving systems of equation, matrix factorization, eigenvalue
- Approximations
- Automatic parallelization by the compiler (e.g. Haskell)

Both

- Parallelization

Section 2

Introduction to parallel processing

Terminology

Hardware

- **Processor / CPU**: Electrical circuit that performs basic operations on external data.
- **Core**: Processing unit.
- **Cluster**: Collection (of machines/cores).

Software

- **Thread**: Sequence of instructions in a process.
- **Process**: Running instance of `R`.
- **Socket**: Duplicated process, uses duplicated memory.
- **Fork**: Duplicated process, uses copy-on-write mechanism.

Language update

~~Master/slave~~ main/worker, parent/child, chief/worker. . .

Types

Socket

- Works on any system and on clusters
- Uses more resources
- Little slower

Forking

- Single machine
- Unix-like operating systems e.g. Linux distributions and macOS
- Uses less resources
- Little faster

Brief history

- `snow` (Simple Network of Workstations) (2003) with `doSNOW`
- `multicore` (2009) with `doMC` (uses forking)

Wrappers:

- `parallel` (2011) with `doParallel`
- `future` (2015) with `dofuture`

More details at
https://cran.r-project.org/web/views/HighPerformanceComputing.html

Where to parallelize

When all tasks are independent, so-called embarrassingly parallel problems (or perfectly, delightfully, pleasingly)

- Monte-Carlo simulations
- Numerical integration
- Computer graphics
- Brute-force searches in cryptography
- Search of hyper parameters in machine learning
- Genetic optimization algorithms
- Cross-validation
- Random forests

Non-embarrassingly parallel problems are more difficult to parallelize.

Should I parallelize? (1)

Time required to write the code & overhead & resources v.s. speedup.

Amdahl's law (1967)

Maximum speedup in latency (inverse of task speed)

$$S(N,p) = \frac{T}{T_{N \text{ cores}}} = \frac{(1-p)T + pT}{(1-p)T + \frac{pT}{N}} = \frac{1}{1-p+\frac{p}{N}}$$

- S = speedup of the task's latency
- N = cores
- p = % task benefiting from parallelization
- T = time with 1 core

Example

If 50% of a problem is sped up ($p = 0.5$) by a factor of 10 ($N = 10$), then the maximum speedup is $S(2, 0.5) = 1.82$.

Should I parallelize? (2)

Gustafson's law (1988)

$$S(N,p) = \frac{T}{T_{N \text{ cores}}} = \frac{(1-p)T + NpT}{(1-p)T + \frac{NpT}{N}} = 1 + p(N-1)$$

- S = speedup of the task's latency
- N = cores
- p = % task benefiting from parallelization
- T = time with 1 core before parallelization

Assumes that the parallelizable work is multiplied by N when parallelized.

Example

If 50% of a problem is sped up ($p = 0.5$) by a factor of 10 ($N = 10$), then the maximum speedup is $S(2, 0.5) = 5.5$.

Should I parallelize? (3)

Sun-Ni's law (1990), memory-bounded speedup, simplified

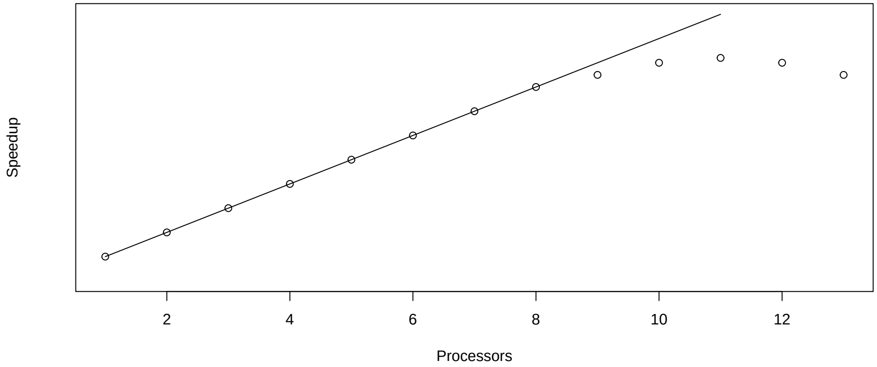
$$S(N,p) = \frac{T}{T_{N\text{ cores}}} = \frac{(1-p)T + g(N)pT}{(1-p)T + \frac{g(N)pT}{N}} = \frac{(1-p) + g(N)p}{(1-p) + \frac{g(N)p}{N}}$$

- S = speedup of the task's latency
- N = cores
- p = % task benefiting from parallelization
- T = time with 1 core before parallelization

Assumes that the parallelizable work is multiplied by $g(N)$ when parallelized.

Parallel slowdown

Non-embarrassingly parallel tasks require communication between processes, which slows down the program.

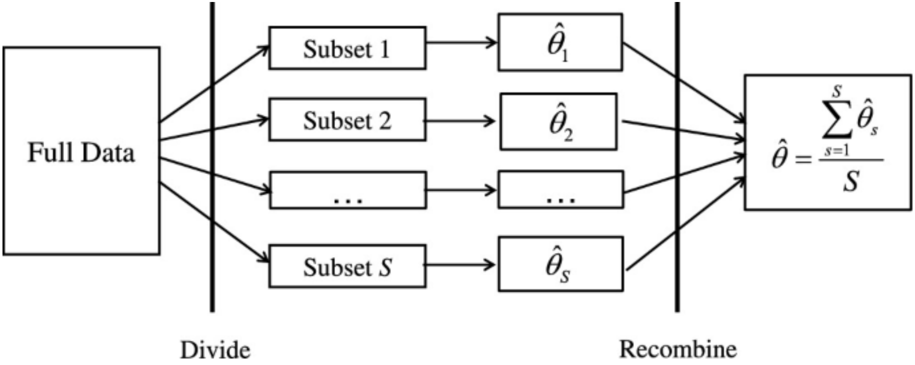


Section 3

Embarassingly parallel problem

Big Data

Divide and recombine (Guha et al. 2012)

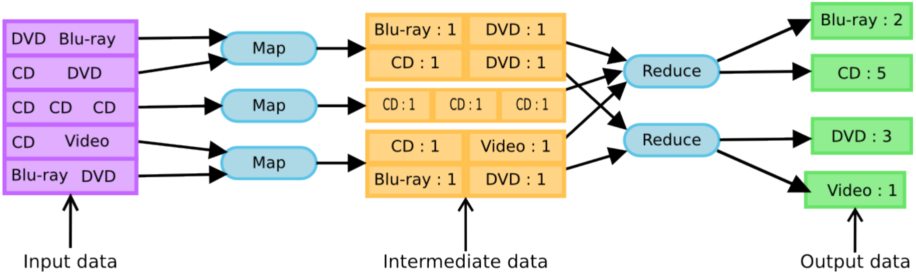


Section 4

Non-embarassingly parallel problems

Big Data

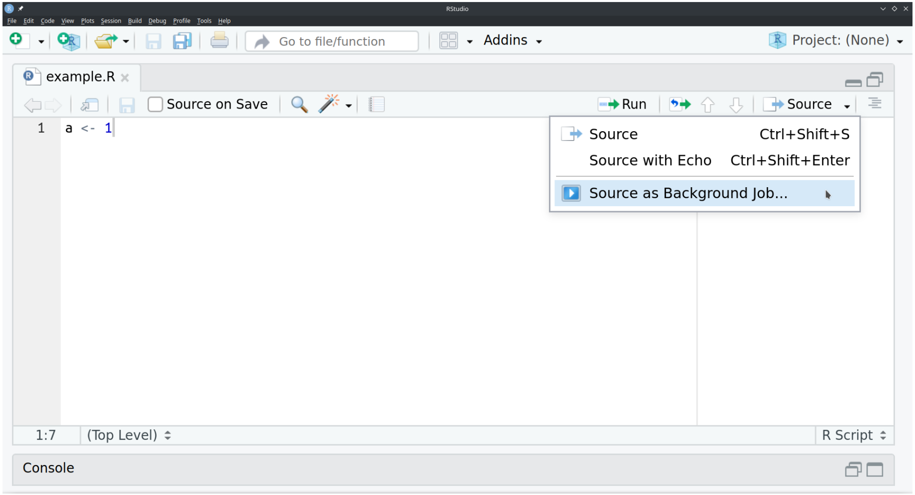
Map Reduce (Dean and Ghemawat 2008)



Section 5

Examples

Natively on RStudio



```
a <- 1
```

Source Ctrl+Shift+S
Source with Echo Ctrl+Shift+Enter
Source as Background Job...
1:7
(Top Level)
R Script
Console

`future`

No worries about

- exporting variables (`clusterExport`)
- packages (`clusterEvalQ`)
- which `apply` function to use (`parLapply`, `mclapply`)
- which parallel back-end to use (`snow`, `multicore`, etc)
- which operating system (Windows, macOS, Linux)

future basics (1)

```
a <- sum(1:100) # Sequential
a
```

```
## [1] 5050
```

```
library(future)
plan(multisession)
# Method 1
fb <- future({ sum(1:50) })
fc <- future( sum(51:100) )
aa <- value(fb) # wait until future is resolved
aa <- aa + value(fc)
aa
```

```
## [1] 5050
```

```
# Method 2
fb %<-% { sum(1:50) }
fc %<-% sum(51:100)
# fc %<-% 1 + 1
aaa <- fb + fc
aaa
```

```
## [1] 5050
```

`future` basics (2)

`apply` and `map` functions

```r
library(future.apply)
plan(multisession)
a <- future_lapply(1:5, sum)
identical(a,
          lapply(1:5, sum))
```

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

```r
# future_replicate()
# future_sapply()
# future_apply()

library(furrr) # futurized `purrr` package
# future_map()
# future_map2()
# future_modify()
```

future Nested loops

```r
library(future)
library(listenv)
x <- listenv()
plan(list(multisession, sequential))
for (i in 1:3) {
  x[[i]] %<-% {

    y <- listenv()
    for (j in 1:3) {
      y[[j]] %<-% {
        return(c(Sys.getpid(), 10*i + j))
      }
    }
    return(y)

  }
}
unlist(x)
```

```
##  [1] 22664    11 22664    12 22664    13 15980    21 15980    22 15980    23
## [13] 17516    31 17516    32 17516    33
```

foreach

Works with/out any parallel computation back-end

```
library(foreach)
pids <- foreach(i = 1:2, .combine = "c") %do% {
  return(Sys.getpid())
}
pids
```

```
## [1] 16876 16876
```

foreach (2)

```r
library(doFuture)
registerDoFuture()
plan(multisession(workers = 2))
pids <- foreach(i = 1:2, .combine = "c") %dopar% {
  return(Sys.getpid())
}
pids
```

```
## [1] 20612  3184
```

foreach Nested loops

Usually, parallelize the outer loop.

```r
registerDoFuture()
plan(list(tweak(multisession, workers = 3),
          tweak(multisession, workers = 2)))
a <- foreach(i = 1:5) %dopar% {
  a <- foreach(j = 1:5) %dopar% {
    return(0)
  }
  return(0)
}
plan(multisession(workers = 2))
a <- foreach(i = 1:5) %:% {
  a <- foreach(j = 1:5) %dopar% {
    return(0)
  }
  return(0)
}
```

foreach Chunking tasks

Each future will process chunk.size elements (on average).

```
registerDoFuture()
plan(multisession(workers = 2))
results <- foreach(i = 1:10,
                   .options.future = list(chunk.size = 3),
                   .combine = "c") %dopar% {
  return(Sys.getpid())
}
results
```

```
##  [1] 17820 17820 17820  5800  5800 17820 17820  5800  5800  5800
```

foreach Randomness

Randomness is a problem.

```r
set.seed(1)
registerDoFuture()
plan(multisession(workers = 2))
results <- foreach(i = 1:2, .combine = "c") %dopar% {
  return(rnorm(1))
}
results
```

```
## [1]  1.2335855 -0.2061181
```

```r
set.seed(1)
rnorm(1)
```

```
## [1] -0.6264538
```

foreach Randomness solution

```r
library(doRNG)
registerDoFuture()
plan(multisession, workers = 2)
results <- foreach(i = 1:2, .combine = "c") %dorng% {
  return(rnorm(1))
}
results[2]
```

```
## [1] -0.1581411
```

```r
.Random.seed <- attr(results, "rng")[[2]]
rnorm(1)
```

```
## [1] -0.1581411
```

Monitoring

```r
library(mailR)
myemail <- "riw011@uib.no"
# Email
message <- "May the force be with you"
send.mail(from = myemail,
          to = myemail,
          subject = "Hello there",
          body = message,
          smtp = list(host.name = "smtp.uib.no",
                      user.name = myemail,
                      passwd = mypassword))
# Online service
system(paste0('curl -d "', message,'" ntfy.sh/aaaa'))
```

Monitoring with progress bar

```r
library(doFuture)
library(progressr)
registerDoFuture()
plan(multisession(workers = 2))

# Monitoring this function
aa <- function() {
  p <- progressor(along = 1:10)
  foreach(i = 1:10) %dopar% {
    Sys.sleep(2)
    p() # Increment progress bar
    return(NULL)
  }
  return(NULL)
}

# Method 1
handlers(global = TRUE)
aa()

# Method 2
handlers(global = FALSE)
with_progress({
  aa()
})
```

Monitoring

Laplacian 2D

Race condition / concurrent writing

```r
library(doFuture)
registerDoFuture()
plan(multisession, workers = 11)
a <- foreach(i = 1:1000) %dopar% {
  write("Dude", file = "log.txt", append = TRUE)
  write("where", file = "log.txt", append = TRUE)
  write("is", file = "log.txt", append = TRUE)
  write("my", file = "log.txt", append = TRUE)
  write("car", file = "log.txt", append = TRUE)
  return(NA)
}
fpeek::peek_tail("log.txt", n = 50, intern = TRUE)
```

```
##  [1] "r"               ""                "iscar"           "Dude"
##  [5] "my"              "Dude"            ""                "wherecarhere"
##  [9] ""                "car"             "isdeis"          "Dude"
## [13] "ere"             "my"              ""                "is"
## [17] "ere"             "carcar"          "is"              ""
## [21] "DudeDude"        ""                "car"             ""
## [25] "wherewherecar"   ""                "Dude"            "is"
## [29] "iswhere"         "mymy"            "is"              "car"
## [33] "carmy"           ""                "Dude"            "car"
## [37] "where"           "Dude"            "is"              "where"
## [41] "my"              "is"              "car"             "my"
## [45] "car"             "Dude"            "where"           "is"
## [49] "my"              "car"
```

Race condition solution

`filelock` (does not work well) or `flock` package

```
library(doFuture)
registerDoFuture()
plan(multisession, workers = 11)
a <- foreach(i = 1:1000, .packages = "flock") %dopar% {

  mylock <- lock("lockfile")
  write("Hasta", file = "log2.txt", append = TRUE)
  write("la", file = "log2.txt", append = TRUE)
  write("vista", file = "log2.txt", append = TRUE)
  write("baby", file = "log2.txt", append = TRUE)
  unlock(mylock)

  return(NA)
}
fpeek::peek_tail("log2.txt", n = 50, intern = TRUE)
```

```
##  [1] "vista" "baby"  "Hasta" "la"    "vista" "baby"  "Hasta" "la"    "vista"
## [10] "baby"  "Hasta" "la"    "vista" "baby"  "Hasta" "la"    "vista" "baby"
## [19] "Hasta" "la"    "vista" "baby"  "Hasta" "la"    "vista" "baby"  "Hasta"
## [28] "la"    "vista" "baby"  "Hasta" "la"    "vista" "baby"  "Hasta" "la"
## [37] "vista" "baby"  "Hasta" "la"    "vista" "baby"  "Hasta" "la"    "vista"
## [46] "baby"  "Hasta" "la"    "vista" "baby"
```

Section 6

Parallel computing at scale

Parallel computing at scale

Distributed computing

- Seti@Home with BOINC software
- Folding@Home, about protein folding
- Fold.it puzzle video game

Apache Hadoop

Google's Map-Reduce

Apache Spark

More efficient on smaller but still massive data sets

More efficient tools

- Schedulers: SLURM
- MPI (Message Programming Interface) with package `rmpi` is de-facto standard
- Managing shared memory processes with e.g. POSIX Threads or OpenMP

e.g. (de Vicente and Rodriguez 2005) uses specialized tools to manage a dynamic cluster of non-dedicated workstations

Section 7

References

References

Dean, Jeffrey, and Sanjay Ghemawat. 2008. "MapReduce: Simplified Data Processing on Large Clusters." *Communications of the ACM* 51 (1): 107–13. https://doi.org/10.1145/1327452.1327492.

Guha, Saptarshi, Ryan Hafen, Jeremiah Rounds, Jin Xia, Jianfu Li, Bowei Xi, and William S. Cleveland. 2012. "Large Complex Data: Divide and Recombine (D&R) with RHIPE." *Stat* 1 (1): 53–67. https://doi.org/10.1002/sta4.7.

Vicente, Angel de, and Nayra Rodriguez. 2005. "Big Science with a Small Budget: Non-embarrassingly Parallel Applications in a Non-Dedicated Network of Workstations." arXiv. https://doi.org/10.48550/ARXIV.CS/0510094.