Advanced R Programming - Lecture 6

Krzysztof Bartoszek
(slides by Leif Jonsson and Måns Magnusson)

Linköping University

krzysztof.bartoszek@liu.se

20 September 2017

Today

Questions since last time?

Writing fast code

Speed is important!
(do not forget memory)

Writing fast code

Speed is important!
(do not forget memory)

Time to write code

Writing fast code

Speed is important!
(do not forget memory)

Time to write code
Time to maintain (understand) code

Writing fast code

Speed is important!
(do not forget memory)

Time to write code
Time to maintain (understand) code
Time to execute code

Old Adage About Software

"You can have it Good, Fast, Cheap. Pick any two."

Performance

1. Performance
2. Complexity

Complexity affects performance

Computational complexity

Theoretical worst case
(but what about average case?)

Big-Oh notation

Basic operations

Relationship: operations to problem size

Big Oh

"How fast does a function grow?"

$$f(n) = O(g(n)) \quad \text{or} \quad f(n) \in O(g(n))$$

$$\exists_{C>0}\ \exists_{N_0\in\mathbb{N}}\ \forall_{\mathbb{N}\ni n>N_0}\ |f(n)| \leq C * |g(n)|$$

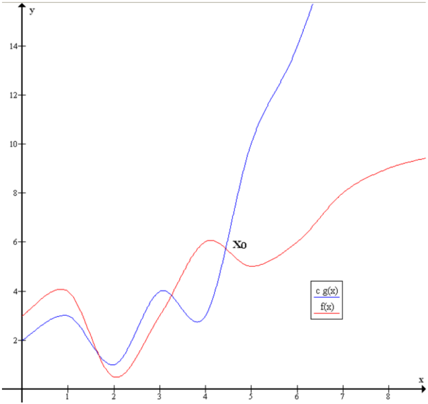
or

$$\limsup_{n\to\infty} \frac{|f(n)|}{|g(n)|} < \infty$$

n number of operations

f does not (up to a scaling constant) grow faster than g

Big Oh



https://en.wikipedia.org/wiki/Big_O_notation

Big Oh

Example

$$f(n) = n^2 + 100n + 100$$

Big Oh

Example

$$f(n) = n^2 + 100n + 100$$

$$f(n) = O(n^2)$$

Other Oh

$$f = o(g) \quad \forall_{C>0}\exists_{N_0\in\mathbb{N}}\forall_{\mathbb{N}\ni n>N_0}|f(n)| \le C|g(n)| \quad \lim_{n\to\infty}\frac{|f(n)|}{|g(n)|} = 0$$

$$f = O(g) \quad \exists_{C>0}\exists_{N_0\in\mathbb{N}}\forall_{\mathbb{N}\ni n>N_0}|f(n)| \le C|g(n)| \quad \limsup_{n\to\infty}\frac{|f(n)|}{|g(n)|} < \infty$$

$$f = \omega(g) \quad \forall_{C>0}\exists_{N_0\in\mathbb{N}}\forall_{\mathbb{N}\ni n>N_0}|f(n)| \ge C|g(n)| \quad \lim_{n\to\infty}\frac{|f(n)|}{|g(n)|} = \infty$$

$$f = \Omega(g) \quad \exists_{C>0}\exists_{N_0\in\mathbb{N}}\forall_{\mathbb{N}\ni n>N_0}f(n) \ge Cg(n) \quad \liminf_{n\to\infty}\frac{f(n)}{g(n)} > 0$$

$$f = \Theta(g) \quad f = O(g) \text{ and } f = \Omega(g)$$

$$f \sim g \quad \lim_{n\to\infty}\frac{f(n)}{g(n)} = 1$$

Complexities (the data size is a lower bound)

Big Oh	**Name**	**Example, optimal**
$O(1)$	constant	assignments, $O(1)$
$O(\log N)$	logarithmic	binary search (sorted input), $O(\log N)$
$O(N)$	linear	max., $O(N)$
$O(N \log N)$	log–linear	sorting, $O(N \log N)$
$O(N^2)$	quadratic	naive vector-matrix mult., preprocessing
$O(N^3)$	cubic	naive matrix inversion, $O(n^{2.373})$
$O(N^3)$	cubic	naive matrix-matrix mult., $O(n^{2.373})$
$O(N^c)$	polynomial	
$O(c^n)$	exponential	brute force cracking of password, **???**

Quicksort: $O(N^2)$ worst case, but $O(N \log N)$ on average

Determine complexity

```
statement 1
statement 2
...
statement c
```

$O(1)$

Determine complexity

```
if(a)
  statement a
else
  statement b
```

max(O(a),O(b))

Determine complexity

```
for(i in 1:N)
  statement i
```

O(n)

Determine complexity

```
for(i in 1:N)
  for (j in 1:M)      O ?
    statement i,j
```

Determine complexity

```
for(i in 1:N)
  for (j in 1:M)
    statement i,j
```

$O(N * M)$

Determine complexity

```
for(i in 1:N)
  g(i)
```

$g(n) = O(n^2)$

$O(n^3)$

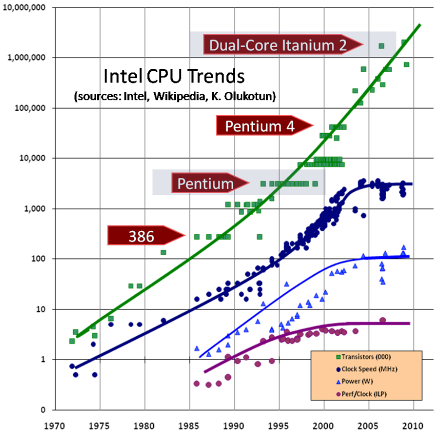
What is parallelism?

Multiple cores

Each core work with its own part

Cores can exchange information

Why parallelism?



http://www.gotw.ca/publications/concurrency-ddj.htm

Why parallelism?

Single core limits

Handling larger data

Solving problems faster

More and more important

Is there any **but** . . . ?

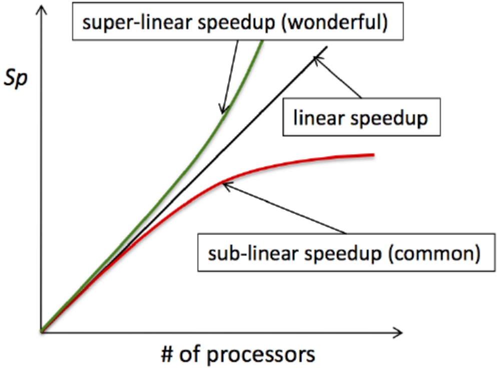
Types of parallelism

Multicore systems

Distributed systems

Graphical processing units (GPU)

Speedup



https://portal.tacc.utexas.edu/c/document_library/get_file?uuid=e05d457a-0fbf-424b-87ce-c96fc0077099&groupId=13601

Theoretical limits

Strong scaling: Amdahl's law

Deals with *fixed problem size, increasing resources*

Weak scaling: Gustafsons law

Deals with *increasing size problem along with increasing resources*

Amdahl's law

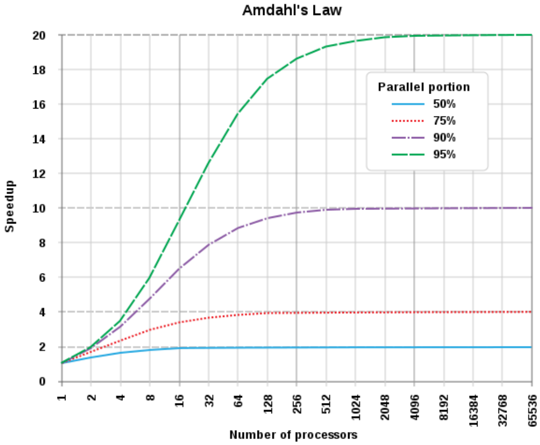
$$S_p = \frac{1}{f_s + \frac{f_p}{P}}$$

Where:

f_s = serial fraction of code
f_p = parallel fraction of code
P = number of cores

For a *fixed size problem*!

Amdahl's law



https://en.wikipedia.org/wiki/Amdahl's_law

Gustafsons law

$$S_p = P - \alpha * (P - 1)$$

Where:

α = the largest non-parallelizable fraction of any parallel process
P = number of cores

Practical problems

Costs of parallelism

communication

load balancing

scheduling

fine-grained vs embarrassingly parallel

Donald E. Knuth on Optimization

Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered.
- Donald E. Knuth

Performance

Depends on many things

1. Code
2. Complexity
3. Compiler
4. Hardware
5. Language

If you don't measure, you don't optimize!

How to optimize

0. Choose optimal algorithm
1. Write code that works with accompanying test suite
2. Profile your code for bottlenecks
3. Try to eliminate the bottle necks
4. Redo 2-3 until fast enough

`proc.time()` is a basic starting tool

Profiling

```
Rprof(tmp <- tempfile(),
  line.profiling = TRUE,
  memory.profiling = TRUE)
test_data <- pxweb::get_pxweb_data(
   url =
     "http://api.scb.se/OV0104/v1/doris/sv/ssd/BE/BE0101
                                    /BE0101A/BefolkningNy",
   dims = list(Region = c('*'),
     Civilstand = c('*'),
     Alder = c('*'),
     Kon = c('*'),
     ContentsCode = c('*'),
     Tid = as.character(1970)),
   clean = TRUE)
Rprof()
summaryRprof(tmp, lines = "show", memory = "both")
```

Profiling

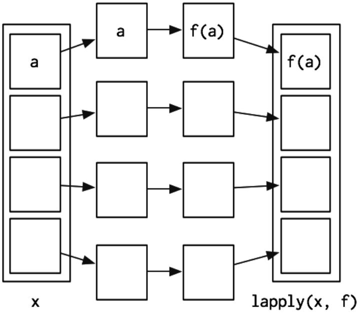
```
$by.self
                               self.time self.pct total.time total.pct mem.total
get_pxweb_data.R#102                1.96     39.2       1.96      39.2     579.2
get_pxweb_data_internal.R#42        1.16     23.2       1.16      23.2     405.0
get_pxweb_data.R#56                 0.52     10.4       0.52      10.4      31.3
get_pxweb_data.R#80                 0.38      7.6       0.38       7.6      29.1
get_pxweb_data.R#82                 0.32      6.4       0.32       6.4      40.7
get_pxweb_data_internal.R#48        0.26      5.2       0.26       5.2      73.2
get_pxweb_data_internal.R#74        0.26      5.2       0.26       5.2      29.8
get_pxweb_data.R#83                 0.08      1.6       0.08       1.6      17.2
api_catalogue.R#75                  0.02      0.4       0.02       0.4       0.0
get_pxweb_data_internal.R#44        0.02      0.4       0.02       0.4      12.6
get_pxweb_data_internal.R#71        0.02      0.4       0.02       0.4      16.0
```

Improvements

0. Optimal data structure and algorithm
1. Look for existing solutions
2. Do less work
3. Vectorise
0. Optimal data structure and algorithm
4. Parallelize
0. Optimal data structure and algorithm
5. Avoid copies

Parallelism in R

Based on lapply()



(H. Wickham, Advanced R, p. 201)

parallel package

Two approaches:

1. mclapply()
2. parLapply()

mclapply()

Pros

Simple to use

Low overhead (startup)

Cons

Does not work on Windows

Only multi core

parLapply(type="psock")

Pros

Works everywhere

Good for testing/developing

Cons

Slow on multiple nodes

parLapply(type="mpi")

Pros

Good for multiple computers Good for production

Cons

Can be used interactively Needs Rmpi package

Example

https://github.com/STIMALiU/AdvRCourse/blob/master/Code/parallel_example.R

Parallel code example

Rcpp

Using C++ code in R

Need C++ compiler (look `http://adv-r.had.co.nz/Rcpp.html`)

Often called interfacing

Similar can be done with Java and Fortran

Extremely fast!

But just handle bottlenecks!

Fibonacci

$$f(n) = \begin{cases} n, & \text{if } n{<}2 \\ F(n-1) + F(n-2), & \text{otherwise} \end{cases}$$

Fibonacci R

```
fr <- function(n) {
  if (n < 2) return(n)
  f(n-1) + f(n-2)
}
```

```
system.time(fr(30))
user   system elapsed
2.246    0.171    2.451
```

Fibonacci C++

```
library(Rcpp)

cppFunction(code = '
  int fcpp(int n) {
    if (n < 2) return(n);
    return(fcpp(n-1) + fcpp(n-2));
  }
')

system.time(fcpp(30))
user        system      elapsed
0.007000000 0.000000000 0.006999999
```

Memoization

A simple optimization technique
Example of a general technique in optimization of trading memory for computation

Memoization stores (caches) results of function calls

If called again, returns old value

Depends on functional programming

Memoise in R

```
> library(memoise)
> a <- function(x) runif(1)
> replicate(3, a())
[1] 0.6709919 0.3490709 0.4772027
> b <- memoise(a)
> replicate(3, b())
[1] 0.1867441 0.1867441 0.1867441
```

Memoise in R

```
> c <- memoise(function(x) {Sys.sleep(1); runif(1)})
> system.time(print(c()))
[1] 0.7816399
user  system elapsed
0.003   0.004   1.001
> system.time(print(c()))
[1] 0.7816399
user  system elapsed
0.001   0.000   0.000
> forget(c)
[1] TRUE
> system.time(print(c()))
[1] 0.9234995
user  system elapsed
0.003   0.004   1.001
```

The End... for today.

Questions?

See you next time!