

# Advanced R Programming - Lecture 6

Leif Jonsson

Linköping University

*leif.jonsson@ericsson.com*

*leif.r.jonsson@liu.se*

September 30, 2016

# Today

Performant Code

Computational complexity

Parallelism

Improving R code

Parallelism in R

Rcpp

Memoization

# Questions since last time?

# Writing fast code

Speed is important!

# Writing fast code

Speed is important!

Time to write code

# Writing fast code

Speed is important!

Time to write code  
Time to maintain (understand) code

# Writing fast code

Speed is important!

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...

# Performance

1. Performance
2. Complexity

Complexity affects performance...

...but performance does'nt affect complexity

# Computational complexity

Theoretical worst case

Big-Oh notation

Basic operations

Relationship: operations to problem size

# Big Oh

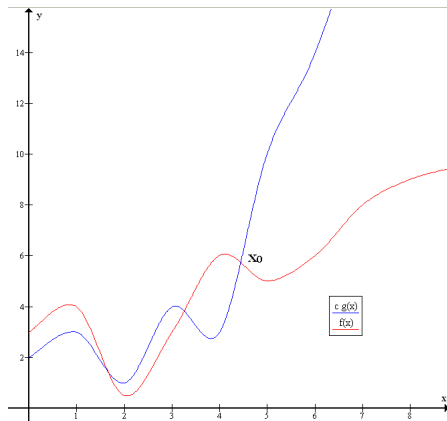
"How fast does a function grow?"

$$f(n) = O(g(n))$$

$$|f(n)| \leq C * |g(n)| \forall n > X_0$$

n    number of operations

# Big Oh



# Big Oh

## Example

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

# Big Oh

## Example

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

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

# Complexities

Big Oh	Name	Example
$O(1)$	constant	assignments
$O(\log(N))$	logarithmic	binary search (of sorted input)
$O(N)$	linear	max
$O(N^2)$	quadratic	naive vector-matrix mult.
$O(N^c)$	polynomial	naive matrix-matrix mult.
$O(c^n)$	exponential	brute force cracking of password



# 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)    O(N * M)
    statement i,j
```

# Determine complexity

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

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

$$O(n^3)$$

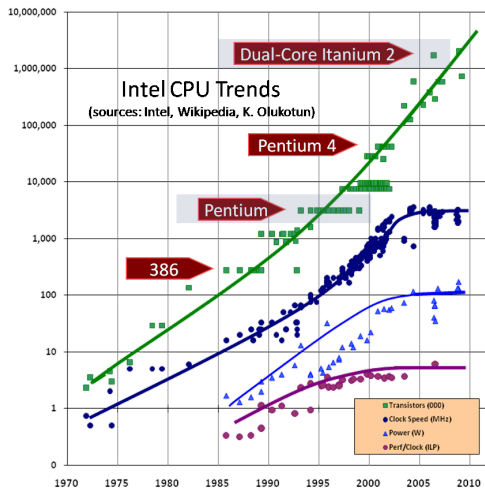
# What is parallelism?

Multiple cores

Each core work with its own part

Cores can exchange information

# Why parallelism?





# Why parallelism?

Single core limits

Handling larger data

Solving problems faster

More and more important

# Types of parallelism

Multicore systems

Distributed systems

Graphical processing units (GPU)

# Speedup

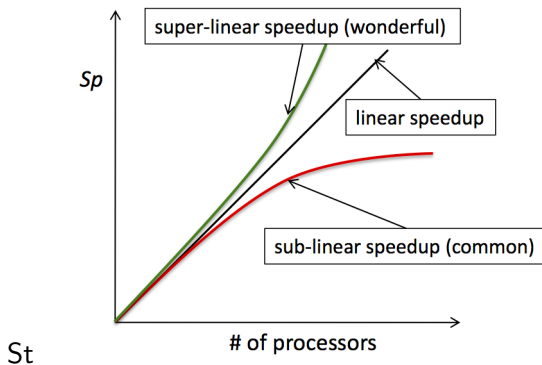


Figure: source

# Theoretical limits

Strong scaling: Amdahl's law

Weak scaling: Gustafsons law

# 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

# Amdahl's law

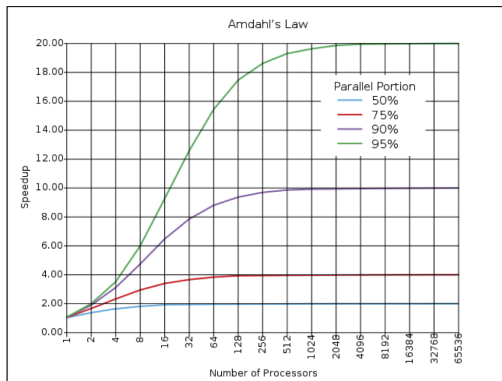


Figure: source

# Gustafsons law

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

Where:

$\alpha$  = 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



# Practical problems

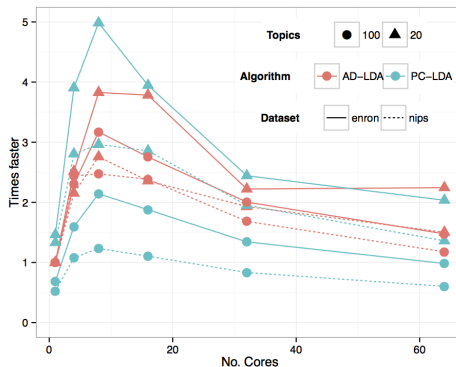


Figure: source

# 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

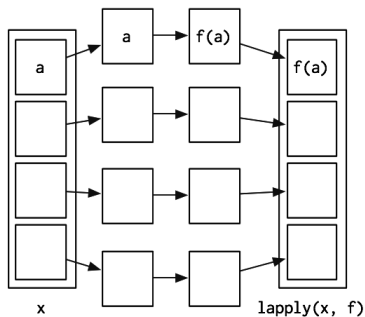
1. Write code that works with accompanying test suite
2. Profile your code for bottlenecks
3. Try to eliminate the bottle necks
4. Redo 2-3 until fast enough

# Improvements

1. Look for existing solutions
2. Do less work
3. Vectorise
4. Parallelize
5. Avoid copies
6. Find smarter algorithms

# Parallelism in R

Based on `lapply()`



# 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

example

# Rcpp

Using C++ code in R

Need C++ compiler (look here)

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

Store results of function calls

If called again, returns old value

Depend 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!