Going Further with the parallel quicksort

Benjamin Cathelineau

26/11/2021

Modifying the shell script in order to acquire metadata about the condition when the experiment was started

As suggested, I added a lot of metadata, like the number of processes, the cpu and ram usage…

I looked on the internet to find a way to extract most of theses metadata.

```
cat scripts/run_benchmarking2.sh
```

```
##
## OUTPUT_DIRECTORY=data/`hostname`_`date +%F`
## mkdir -p $OUTPUT_DIRECTORY
## OUTPUT_FILE=$OUTPUT_DIRECTORY/measurements_`date +%R`.csv
##
## touch $OUTPUT_FILE
##
## # scheduler settings
## sudo renice --priority -19 --pid $BASHPID > /dev/null
## echo "Size, TimeSeq, TimePar,TimeBuiltIn, CPU_USAGE,RAM_USAGE,STORAGE_USAGE,N_PROCESS, ON_BATTERY,TE
## for i in $(seq 1 5); do
##     for j in $(seq 0 1000000 10000000) ; do
##         echo -n "$j" >> $OUTPUT_FILE
##         ./src/parallelQuicksort $j >> $OUTPUT_FILE
##
##
##         # Collection metadata
##         # cpu usage https://unix.stackexchange.com/questions/69167/bash-script-that-print-cpu-usage-d
##         echo -n  ,`top -b -n1 | grep "Cpu(s)" | awk '{print $2 + $4}'` >> $OUTPUT_FILE
##
##         # ram usage
##  FREE_DATA=`free -m | grep Mem`
##  CURRENT=`echo $FREE_DATA | cut -f3 -d' '`
##  TOTAL=`echo $FREE_DATA | cut -f2 -d' '`
##  echo -n ,$(echo "scale = 2; $CURRENT/$TOTAL*100" | bc)>> $OUTPUT_FILE
##  # hdd usage
##  echo -n ,`df -lh | awk '{if ($6 == "/") { print $5 }}' | head -1 | cut -d'%' -f1`>> $OUTPUT_FILE
##  # number of processes
##  echo -n ,$(ps aux | wc -l)>> $OUTPUT_FILE
##  # on battery or not
##  echo -n , >> $OUTPUT_FILE
##  upower -i `upower -e | grep 'BAT'` |grep 'state' | sed "s/state://g" | tr -d '\n' | xargs | tr -d '
##  # Temperature https://askubuntu.com/questions/779819/cpu-temperature-embedded-in-bash-command-prompt
##  echo -n  ,$(sensors | grep -oP 'CPU Die Core Temp.*?\+\K[0-9.]+')>> $OUTPUT_FILE
```

```
##
##  # cpu governor https://unix.stackexchange.com/questions/182696/how-to-get-current-cpupower-governor
##  echo ,$(cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_governor)>> $OUTPUT_FILE
##     done
## done
```

Modyfing the C program in order not to have to deal with perl

Now, the C program directly outputs a csv file, because I don't know Perl and I don't have time to learn it. The printing was changed.

```
  printf(", %lf ", diff);
```

Customizing the linux scheduler settings in order to increase our test priorities

At the start of the shell script, I run this command:

```
sudo renice --priority -19 --pid $BASHPID > /dev/null
```

This changes the scheduler setting so that our test is favored. See the man page. In short a lower niceness is better for the process. A higher niceness is worse.

Analysing the new data, with confidence intervals

```
library(dplyr)
```

```
##
## Attachement du package : 'dplyr'

## Les objets suivants sont masqués depuis 'package:stats':
##
##     filter, lag

## Les objets suivants sont masqués depuis 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
library(ggplot2)
library(Rmisc)
```

```
## Le chargement a nécessité le package : lattice

## Le chargement a nécessité le package : plyr

## ------------------------------------------------------------------------------

## You have loaded plyr after dplyr - this is likely to cause problems.
## If you need functions from both plyr and dplyr, please load plyr first, then dplyr:
## library(plyr); library(dplyr)

## ------------------------------------------------------------------------------

##
## Attachement du package : 'plyr'

## Les objets suivants sont masqués depuis 'package:dplyr':
##
##     arrange, count, desc, failwith, id, mutate, rename, summarise,
##     summarize
```

```r
library(tidyverse)
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.1 --
## v tibble  3.1.6     v purrr   0.3.4
## v tidyr   1.1.4     v stringr 1.4.0
## v readr   2.1.1     v forcats 0.5.1
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x plyr::arrange()   masks dplyr::arrange()
## x purrr::compact()  masks plyr::compact()
## x plyr::count()     masks dplyr::count()
## x plyr::failwith()  masks dplyr::failwith()
## x dplyr::filter()   masks stats::filter()
## x plyr::id()        masks dplyr::id()
## x dplyr::lag()      masks stats::lag()
## x plyr::mutate()    masks dplyr::mutate()
## x plyr::rename()    masks dplyr::rename()
## x plyr::summarise() masks dplyr::summarise()
## x plyr::summarize() masks dplyr::summarize()
```

```r
df = read.csv("data/fedora_2021-12-01/measurements_14:37.csv") # load the data from the CSV
df
```

```
##        Size  TimeSeq  TimePar TimeBuiltIn CPU_USAGE RAM_USAGE STORAGE_USAGE
## 1         0 0.000000 0.000511    0.000004         9        46            14
## 2   1000000 0.122787 0.180089    0.201578        10        46            14
## 3   2000000 0.258838 0.345271    0.394852        10        46            14
## 4   3000000 0.389309 0.472851    0.614174         9        46            14
## 5   4000000 0.537799 0.643878    0.830870        25        46            14
## 6   5000000 0.713201 0.801830    1.074060        17        46            14
## 7   6000000 0.818768 0.902943    1.297154        11        46            14
## 8   7000000 1.006788 1.042709    1.533210        11        46            14
## 9   8000000 1.181180 1.255406    1.767254        11        46            14
## 10  9000000 1.257162 1.410159    2.031089        12        46            14
## 11 10000000 1.451663 1.634807    2.303282        12        46            14
## 12        0 0.000001 0.000371    0.000003        14        46            14
## 13  1000000 0.125794 0.193168    0.190120        13        46            14
## 14  2000000 0.262169 0.335554    0.421010        19        46            14
## 15  3000000 0.409404 0.505379    0.662281        10        46            14
## 16  4000000 0.567011 0.675450    0.903268        15        46            14
## 17  5000000 0.700046 0.830256    1.126786        18        46            14
## 18  6000000 0.869244 0.938737    1.361411        10        46            14
## 19  7000000 0.985444 1.176316    1.599060        14        46            14
## 20  8000000 1.132755 1.327023    1.788308        17        46            14
## 21  9000000 1.309040 1.492488    2.211324        16        46            14
## 22 10000000 1.430431 1.605811    2.503851        11        46            14
## 23        0 0.000000 0.000505    0.000002        12        46            14
## 24  1000000 0.126942 0.195896    0.190241        13        46            14
## 25  2000000 0.268907 0.338938    0.407934        22        46            14
## 26  3000000 0.423347 0.510146    0.669175        15        46            14
## 27  4000000 0.591148 0.660241    0.886063        16        46            14
## 28  5000000 0.715351 0.807237    1.098474        36        46            14
## 29  6000000 0.838489 0.957679    1.343032        23        46            14
## 30  7000000 0.989882 1.116152    1.673185        10        46            14
## 31  8000000 1.120642 1.242341    1.817161        20        46            14
```

```
## 32  9000000 1.315064 1.401234    1.994647        10         46             14
## 33 10000000 1.407706 1.575461    2.216965         9         46             14
## 34        0 0.000000 0.000372    0.000011        15         46             14
## 35  1000000 0.127388 0.183451    0.197383        21         46             14
## 36  2000000 0.272571 0.335810    0.397416         8         46             14
## 37  3000000 0.391157 0.481363    0.613656        13         46             14
## 38  4000000 0.539626 0.648876    0.837274        11         46             14
## 39  5000000 0.670290 0.789260    1.055956        12         46             14
## 40  6000000 0.811055 0.957940    1.300443        10         46             14
## 41  7000000 0.953955 1.094826    1.538587        11         46             14
## 42  8000000 1.123812 1.281357    1.763244        18         46             14
## 43  9000000 1.276537 1.431894    1.965513         9         46             14
## 44 10000000 1.400998 1.536765    2.238027        21         46             14
## 45        0 0.000000 0.000539    0.000003        18         46             14
## 46  1000000 0.123637 0.185461    0.197273        11         46             14
## 47  2000000 0.256073 0.331379    0.413583         9         46             14
## 48  3000000 0.402250 0.512991    0.608351         9         46             14
## 49  4000000 0.536034 0.651125    0.839474        10         46             14
## 50  5000000 0.671789 0.781475    1.055982        11         46             14
## 51  6000000 0.850489 0.942595    1.321626        14         46             14
## 52  7000000 0.978512 1.096399    1.533914        27         46             14
## 53  8000000 1.152610 1.368908    2.070410        22         46             14
## 54  9000000 1.308921 1.407782    2.117979        11         46             14
## 55 10000000 1.499842 1.915462    2.464082        10         46             14
##    N_PROCESS    ON_BATTERY TEMPERATURE CPU_GOVERNOR
## 1        374 fully-charged        62.2    schedutil
## 2        374 fully-charged        63.0    schedutil
## 3        375 fully-charged        67.0    schedutil
## 4        375 fully-charged        72.2    schedutil
## 5        375 fully-charged        70.5    schedutil
## 6        375 fully-charged        74.2    schedutil
## 7        375 fully-charged        76.0    schedutil
## 8        375 fully-charged        77.0    schedutil
## 9        376 fully-charged        78.5    schedutil
## 10       375 fully-charged        81.2    schedutil
## 11       375 fully-charged        82.0    schedutil
## 12       375 fully-charged        82.0    schedutil
## 13       375 fully-charged        83.2    schedutil
## 14       375 fully-charged        84.8    schedutil
## 15       375 fully-charged        86.2    schedutil
## 16       375 fully-charged        83.8    schedutil
## 17       375 fully-charged        86.8    schedutil
## 18       375 fully-charged        87.5    schedutil
## 19       375 fully-charged        89.0    schedutil
## 20       376 fully-charged        91.8    schedutil
## 21       376 fully-charged        86.8    schedutil
## 22       376 fully-charged        87.2    schedutil
## 23       376 fully-charged        84.8    schedutil
## 24       376 fully-charged        84.8    schedutil
## 25       376 fully-charged        88.8    schedutil
## 26       376 fully-charged        88.0    schedutil
## 27       376 fully-charged        90.5    schedutil
## 28       376 fully-charged        87.8    schedutil
## 29       376 fully-charged        87.8    schedutil
```

```
## 30       376 fully-charged        87.2    schedutil
## 31       376 fully-charged        89.5    schedutil
## 32       376 fully-charged        89.5    schedutil
## 33       376 fully-charged        85.5    schedutil
## 34       376 fully-charged        85.5    schedutil
## 35       376 fully-charged        87.0    schedutil
## 36       376 fully-charged        89.2    schedutil
## 37       376 fully-charged        88.5    schedutil
## 38       376 fully-charged        91.5    schedutil
## 39       376 fully-charged        91.2    schedutil
## 40       376 fully-charged        89.2    schedutil
## 41       376 fully-charged        89.0    schedutil
## 42       376 fully-charged        89.8    schedutil
## 43       377 fully-charged        88.5    schedutil
## 44       378 fully-charged        88.2    schedutil
## 45       378 fully-charged        88.2    schedutil
## 46       377 fully-charged        87.0    schedutil
## 47       377 fully-charged        88.8    schedutil
## 48       377 fully-charged        90.5    schedutil
## 49       377 fully-charged        88.0    schedutil
## 50       377 fully-charged        86.5    schedutil
## 51       378 fully-charged        88.5    schedutil
## 52       377 fully-charged        90.5    schedutil
## 53       377 fully-charged        89.0    schedutil
## 54       378 fully-charged        92.5    schedutil
## 55       380 fully-charged        87.5    schedutil
```

You can see that now there is a lot more metadata for each entry

plot and confidence interval on the time

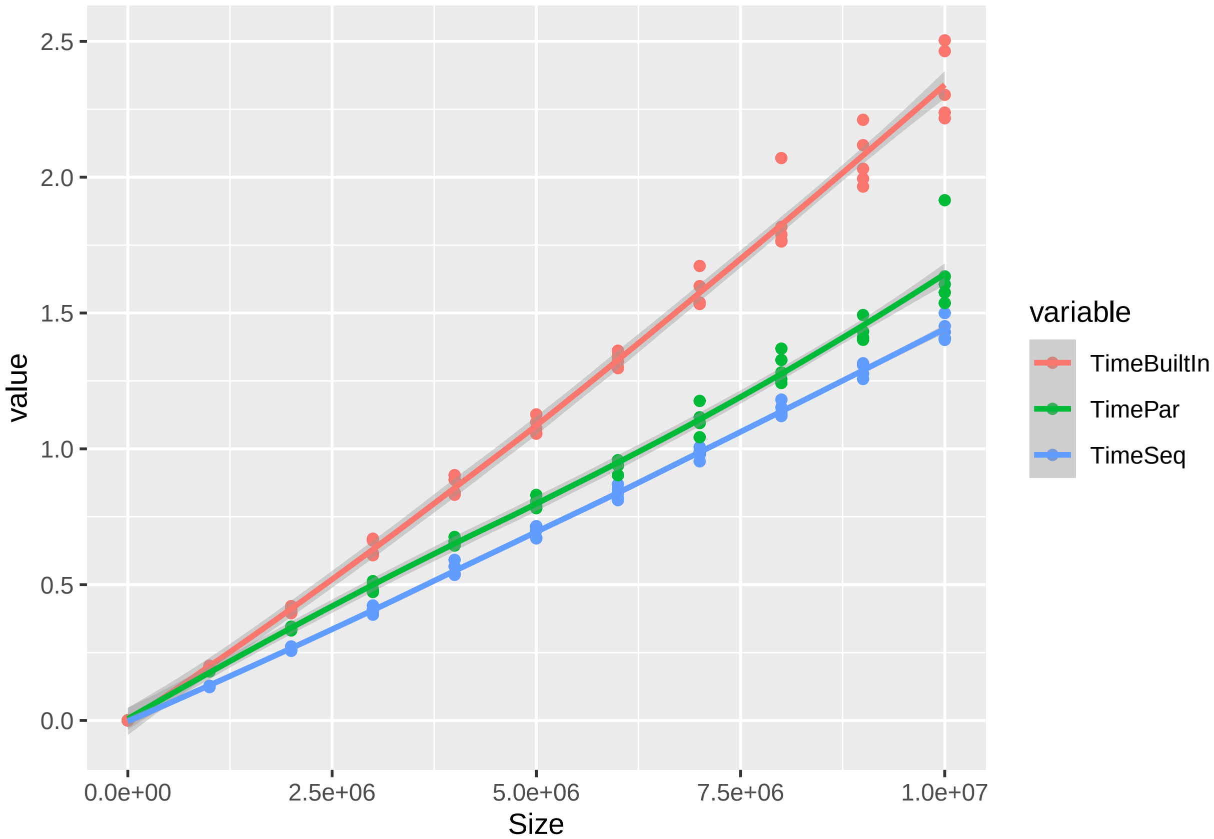
Here I plot the new data and get essentialy the same result as before. I have to do a wierd transformation because of the way the data is now.

```r
# https://www.datanovia.com/en/fr/blog/comment-creer-un-ggplot-contenant-plusieurs-lignes/
df2 <- df %>%
  dplyr::select(Size, TimeSeq, TimePar,TimeBuiltIn) %>%
  gather(key = "variable", value = "value", -Size)

p <- ggplot(df2, aes(x=Size, y = value, color = variable)) + geom_point() +geom_smooth()

print(p)
```

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



Then I try to compute the confidence interval on time for each size and for each algorithm. That's a lot of data.

```
# Intended for a .95 confidence
MY_CI <- function(data){
  data_mean = mean(data)
  std_dev = sd(data)
  final_mult= std_dev/length(data)
  return(c(up= data_mean+(final_mult*2),mean=data_mean,down= data_mean-(final_mult*2)))

}

df %>% group_by(Size) %>% group_map(~ CI(x=.x$TimeSeq,ci=.95))
```

```
## [[1]]
##         upper          mean         lower
##  7.55289e-07  2.00000e-07 -3.55289e-07
##
## [[2]]
##     upper      mean     lower
## 0.1278223 0.1253096 0.1227969
##
## [[3]]
##     upper      mean     lower
## 0.2722683 0.2637116 0.2551549
##
## [[4]]
```

```
##     upper      mean     lower
## 0.4204682 0.4030934 0.3857186
##
## [[5]]
##     upper      mean     lower
## 0.5843593 0.5543236 0.5242879
##
## [[6]]
##     upper      mean     lower
## 0.7213141 0.6941354 0.6669567
##
## [[7]]
##     upper      mean     lower
## 0.8669103 0.8376090 0.8083077
##
## [[8]]
##     upper      mean     lower
## 1.0068237 0.9829162 0.9590087
##
## [[9]]
##    upper     mean    lower
## 1.173365 1.142200 1.111035
##
## [[10]]
##    upper     mean    lower
## 1.324710 1.293345 1.261980
##
## [[11]]
##    upper     mean    lower
## 1.487620 1.438128 1.388636
```

```
df %>% group_by(Size) %>% group_map(~ MY_CI(.x$TimeSeq))
```

```
## [[1]]
##           up         mean         down
## 3.788854e-07 2.000000e-07 2.111456e-08
##
## [[2]]
##        up      mean      down
## 0.1261190 0.1253096 0.1245002
##
## [[3]]
##        up      mean      down
## 0.2664681 0.2637116 0.2609551
##
## [[4]]
##        up      mean      down
## 0.4086907 0.4030934 0.3974961
##
## [[5]]
##        up      mean      down
## 0.5639995 0.5543236 0.5446477
##
## [[6]]
##        up      mean      down
```

```
## 0.7028910 0.6941354 0.6853798
##
## [[7]]
##        up      mean      down
## 0.8470484 0.8376090 0.8281696
##
## [[8]]
##        up      mean      down
## 0.9906179 0.9829162 0.9752145
##
## [[9]]
##       up     mean     down
## 1.152239 1.142200 1.132160
##
## [[10]]
##       up     mean     down
## 1.303449 1.293345 1.283241
##
## [[11]]
##       up     mean     down
## 1.454072 1.438128 1.422184
```

```
df %>% group_by(Size) %>% group_map(~ CI(x=.x$TimePar,ci=.95))
```

```
## [[1]]
##        upper         mean        lower
## 0.0005607236 0.0004596000 0.0003584764
##
## [[2]]
##     upper      mean     lower
## 0.1958966 0.1876130 0.1793294
##
## [[3]]
##     upper      mean     lower
## 0.3437975 0.3373904 0.3309833
##
## [[4]]
##    upper     mean    lower
## 0.519148 0.496546 0.473944
##
## [[5]]
##    upper     mean    lower
## 0.671344 0.655914 0.640484
##
## [[6]]
##     upper      mean     lower
## 0.8253192 0.8020116 0.7787040
##
## [[7]]
##     upper      mean     lower
## 0.9678537 0.9399788 0.9121039
##
## [[8]]
##    upper     mean    lower
## 1.165047 1.105280 1.045514
```

```
##
## [[9]]
##    upper     mean    lower
## 1.360164 1.295007 1.229850
##
## [[10]]
##    upper     mean    lower
## 1.475232 1.428711 1.382191
##
## [[11]]
##    upper     mean    lower
## 1.840906 1.653661 1.466416
```

```
df %>% group_by(Size) %>% group_map(~ MY_CI(.x$TimePar))
```

```
## [[1]]
##           up         mean         down
## 0.0004921768 0.0004596000 0.0004270232
##
## [[2]]
##        up      mean      down
## 0.1902816 0.1876130 0.1849444
##
## [[3]]
##        up      mean      down
## 0.3394544 0.3373904 0.3353264
##
## [[4]]
##        up      mean      down
## 0.5038272 0.4965460 0.4892648
##
## [[5]]
##        up      mean      down
## 0.6608848 0.6559140 0.6509432
##
## [[6]]
##        up      mean      down
## 0.8095201 0.8020116 0.7945031
##
## [[7]]
##        up      mean      down
## 0.9489586 0.9399788 0.9309990
##
## [[8]]
##       up     mean     down
## 1.124534 1.105280 1.086027
##
## [[9]]
##       up     mean     down
## 1.315997 1.295007 1.274017
##
## [[10]]
##       up     mean     down
## 1.443698 1.428711 1.413725
##
```

```
## [[11]]
##       up     mean     down
## 1.713982 1.653661 1.593340
```

```r
df %>% group_by(Size) %>% group_map(~ CI(x=.x$TimeBuiltIn,ci=.95))
```

```
## [[1]]
##        upper         mean        lower
## 9.128245e-06 4.600000e-06 7.175507e-08
##
## [[2]]
##     upper      mean     lower
## 0.2015295 0.1953190 0.1891085
##
## [[3]]
##     upper      mean     lower
## 0.4205594 0.4069590 0.3933586
##
## [[4]]
##     upper      mean     lower
## 0.6702605 0.6335274 0.5967943
##
## [[5]]
##     upper      mean     lower
## 0.9002700 0.8593898 0.8185096
##
## [[6]]
##    upper     mean    lower
## 1.119990 1.082252 1.044513
##
## [[7]]
##    upper     mean    lower
## 1.358966 1.324733 1.290501
##
## [[8]]
##    upper     mean    lower
## 1.651571 1.575591 1.499611
##
## [[9]]
##    upper     mean    lower
## 2.002522 1.841275 1.680029
##
## [[10]]
##    upper     mean    lower
## 2.188603 2.064110 1.939617
##
## [[11]]
##    upper     mean    lower
## 2.508309 2.345241 2.182173
```

```r
df %>% group_by(Size) %>% group_map(~ MY_CI(.x$TimeBuiltIn))
```

```
## [[1]]
##           up         mean         down
## 6.058767e-06 4.600000e-06 3.141233e-06
```

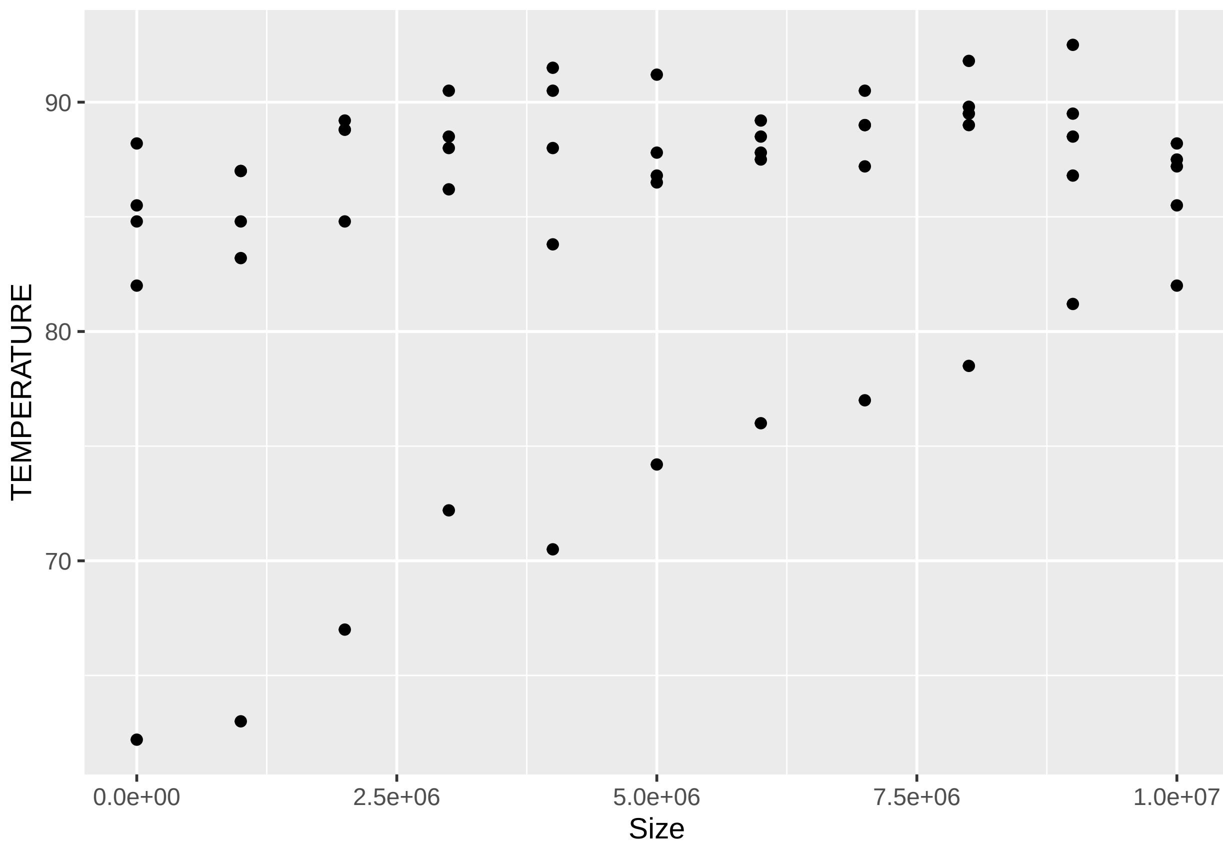
```
##
## [[2]]
##        up      mean      down
## 0.1973197 0.1953190 0.1933183
##
## [[3]]
##        up      mean      down
## 0.4113404 0.4069590 0.4025776
##
## [[4]]
##        up      mean      down
## 0.6453609 0.6335274 0.6216939
##
## [[5]]
##        up      mean      down
## 0.8725593 0.8593898 0.8462203
##
## [[6]]
##       up     mean     down
## 1.094409 1.082252 1.070094
##
## [[7]]
##       up     mean     down
## 1.335761 1.324733 1.313705
##
## [[8]]
##       up     mean     down
## 1.600068 1.575591 1.551114
##
## [[9]]
##       up     mean     down
## 1.893221 1.841275 1.789330
##
## [[10]]
##       up     mean     down
## 2.104216 2.064110 2.024005
##
## [[11]]
##       up     mean     down
## 2.397773 2.345241 2.292709
```

As you can see, I was not able to reproduce the behavior of the `CI` function from the `Rmisc`package. I found the source code for the `CI` function here, and I don't understand what they are doing with the `qt` function. Maybe it's related to what *Arnaud Legrand* said about the sample variance being unreliable. However I'm pretty close to the same confidence interval nonetheless.

Exploiting the metadata

I only had time to look at the temperature.

```
df %>% ggplot(aes(x=Size, y=TEMPERATURE)) + geom_point()
```



I believe that what we see here is the first runs of the test slowly increasing the temperature, and then it mostly stagnates around 80-90c°.

In general, all the metadata should be able to tell us if something goes wrong.

Using the profiler to figure out in what function most of the code spends its time

I added a new entry in the makefile to compile with the right option in order to use the `gprof` profiler

```
cat src/Makefile
```

```
## parallelQuicksort: parallelQuicksort.o
##
##
##
## CFLAGS = -Wall -O3 -pthread -lrt -std=c99
##
## PROFFLAGS= -pg
##
## %: %.o
##  $(CC) $(INCLUDES) $(DEFS) $(CFLAGS) $^ $(LIBS) -o $@
##
## %.o: %.c
##  $(CC) $(INCLUDES) $(DEFS) $(CFLAGS) -c -o $@ $<
##
## clean:
```

```
##  rm -f gmon.out parallelQuicksort profiling *.o *~
##
## profiling:
##  $(CC) $(CFLAGS) $(PROFFLAGS) *.c -o $@
##  ./profiling > /dev/null
##  gprof profiling gmon.out > gprof.txt
```

```
make -C src/ clean
make -C src/ profiling
```

```
## make : on entre dans le répertoire « /home/benjamin/git/M2R-ParallelQuicksort/src »
## rm -f gmon.out parallelQuicksort profiling *.o *~
## make : on quitte le répertoire « /home/benjamin/git/M2R-ParallelQuicksort/src »
## make : on entre dans le répertoire « /home/benjamin/git/M2R-ParallelQuicksort/src »
## cc -Wall -O3 -pthread -lrt -std=c99  -pg *.c -o profiling
## ./profiling > /dev/null
## gprof profiling gmon.out > gprof.txt
## make : on quitte le répertoire « /home/benjamin/git/M2R-ParallelQuicksort/src »
```

```
cat src/gprof.txt
```

```
## Flat profile:
##
## Each sample counts as 0.01 seconds.
##   %   cumulative   self              self     total
##  time   seconds   seconds    calls  us/call  us/call  name
##  80.00      0.32     0.32  1224336     0.26     0.26  partition
##  15.00      0.38     0.06                             compare_doubles
##   5.00      0.40     0.02      797    25.09   426.32  quicksortHelper
##
##  %         the percentage of the total running time of the
## time       program used by this function.
##
## cumulative a running sum of the number of seconds accounted
##  seconds   for by this function and those listed above it.
##
##  self      the number of seconds accounted for by this
## seconds    function alone.  This is the major sort for this
##            listing.
##
## calls      the number of times this function was invoked, if
##            this function is profiled, else blank.
##
##  self      the average number of milliseconds spent in this
## ms/call    function per call, if this function is profiled,
##     else blank.
##
##  total     the average number of milliseconds spent in this
## ms/call    function and its descendents per call, if this
##     function is profiled, else blank.
##
## name       the name of the function.  This is the minor sort
##            for this listing. The index shows the location of
##     the function in the gprof listing. If the index is
##     in parenthesis it shows where it would appear in
```

```
##     the gprof listing if it were to be printed.
##

## Copyright (C) 2012-2021 Free Software Foundation, Inc.
##
## Copying and distribution of this file, with or without modification,
## are permitted in any medium without royalty provided the copyright
## notice and this notice are preserved.
##

##           Call graph (explanation follows)
##
##
## granularity: each sample hit covers 4 byte(s) for 2.50% of 0.40 seconds
##
## index % time    self  children    called     name
##                                                  <spontaneous>
## [1]     85.0    0.00    0.34                 parallelQuicksortHelper [1]
##                 0.02    0.32     797/797         quicksortHelper [2]
##                 0.00    0.00     864/1224336     partition [3]
## -----------------------------------------------
##                              131390             quicksortHelper [2]
##                 0.02    0.32     797/797         parallelQuicksortHelper [1]
## [2]     84.9    0.02    0.32     797+131390  quicksortHelper [2]
##                 0.32    0.00 1223472/1224336     partition [3]
##                              131390             quicksortHelper [2]
## -----------------------------------------------
##                 0.00    0.00     864/1224336     parallelQuicksortHelper [1]
##                 0.32    0.00 1223472/1224336     quicksortHelper [2]
## [3]     80.0    0.32    0.00 1224336         partition [3]
## -----------------------------------------------
##                                                  <spontaneous>
## [4]     15.0    0.06    0.00                 compare_doubles [4]
## -----------------------------------------------
##
##  This table describes the call tree of the program, and was sorted by
##  the total amount of time spent in each function and its children.
##
##  Each entry in this table consists of several lines.  The line with the
##  index number at the left hand margin lists the current function.
##  The lines above it list the functions that called this function,
##  and the lines below it list the functions this one called.
##  This line lists:
##      index   A unique number given to each element of the table.
##      Index numbers are sorted numerically.
##      The index number is printed next to every function name so
##      it is easier to look up where the function is in the table.
##
##      % time  This is the percentage of the ‘total’ time that was spent
##      in this function and its children.  Note that due to
##      different viewpoints, functions excluded by options, etc,
##      these numbers will NOT add up to 100%.
##
##      self    This is the total amount of time spent in this function.
```

```
##
##      children    This is the total amount of time propagated into this
##      function by its children.
##
##      called  This is the number of times the function was called.
##      If the function called itself recursively, the number
##      only includes non-recursive calls, and is followed by
##      a ‘+’ and the number of recursive calls.
##
##      name    The name of the current function.  The index number is
##      printed after it.  If the function is a member of a
##      cycle, the cycle number is printed between the
##      function’s name and the index number.
##
##
##  For the function’s parents, the fields have the following meanings:
##
##      self    This is the amount of time that was propagated directly
##      from the function into this parent.
##
##      children    This is the amount of time that was propagated from
##      the function’s children into this parent.
##
##      called  This is the number of times this parent called the
##      function ‘/’ the total number of times the function
##      was called.  Recursive calls to the function are not
##      included in the number after the ‘/’.
##
##      name    This is the name of the parent.  The parent’s index
##      number is printed after it.  If the parent is a
##      member of a cycle, the cycle number is printed between
##      the name and the index number.
##
##  If the parents of the function cannot be determined, the word
##  ‘<spontaneous>’ is printed in the ‘name’ field, and all the other
##  fields are blank.
##
##  For the function’s children, the fields have the following meanings:
##
##      self    This is the amount of time that was propagated directly
##      from the child into the function.
##
##      children    This is the amount of time that was propagated from the
##      child’s children to the function.
##
##      called  This is the number of times the function called
##      this child ‘/’ the total number of times the child
##      was called.  Recursive calls by the child are not
##      listed in the number after the ‘/’.
##
##      name    This is the name of the child.  The child’s index
##      number is printed after it.  If the child is a
##      member of a cycle, the cycle number is printed
##      between the name and the index number.
```

```
##
##  If there are any cycles (circles) in the call graph, there is an
##  entry for the cycle-as-a-whole.  This entry shows who called the
##  cycle (as parents) and the members of the cycle (as children.)
##  The '+' recursive calls entry shows the number of function calls that
##  were internal to the cycle, and the calls entry for each member shows,
##  for that member, how many times it was called from other members of
##  the cycle.
##

## Copyright (C) 2012-2021 Free Software Foundation, Inc.
##
## Copying and distribution of this file, with or without modification,
## are permitted in any medium without royalty provided the copyright
## notice and this notice are preserved.
##

## Index by function name
##
##    [4] compare_doubles         [3] partition               [2] quicksortHelper
```

We see that most of the time is spent in the `partition` function.