Benchmarking Analysis for a Multi-Threaded Version of Quicksort

Authors: Oussama Oulkaid, Dalia Hareb, Walid Alihaimoud

November, 2021

1. Introduction

The aim of this activity is to analyse the time spent by Quicksort, by pinpointing the parameters that might affect its performance (array size, number of cores on the machine, nature of other applications running at the same time, etc.).

To get started, compile the program by running:

```
make -C src/
```

2. Choices we made

The `run_benchmarking.sh` script have been modified so that the array size samples are chosen to be incremented by the same amount. And to simplify the experimentation process, the Perl script that build the csv file is now being run within the `run_benchmarking.sh`. Also, the txt files are not anymore preserved.

To launch an experiment you need to set the `START_SIZE`, `MAX_SIZE` and `STEP` constants as you want. Then, run:

```
./scripts/run_benchmarking.sh
```

```
# The environment
library(tidyverse)
library(ggplot2)
library(reshape2)
```

3. Experimentation and analysis

Now, let's build the dataframes from csv files and plot the corresponding graphs with ggplot2.

Each experiment is run on a different machine. The goal is to explore the effect of system capabilities on the overall performance (and maybe make conclusions upon the patterns found). In the following plots, we represent obtained samples by small dots and we draw a line between the means of theses samples.

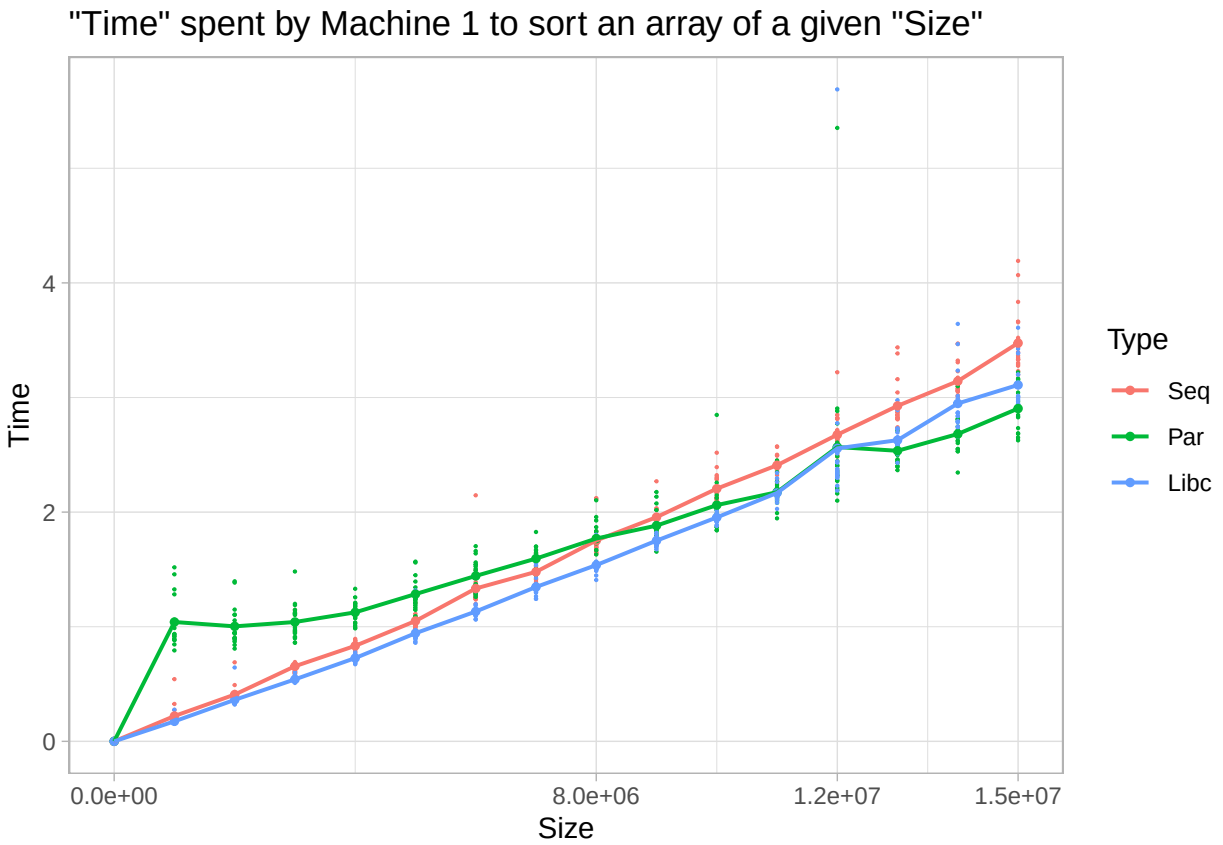
3.1. Machine 1 (Virtual Machine): 4 cores, 1 thread per core, 2370 CPU MHz

Resulting data: `data/in_2021-12-08/measurements_19:36.csv`

```
#parameters: START_SIZE=0, MAX_SIZE=15000000, STEP=1000000
df <- read.csv("data/in_2021-12-08/measurements_19:36.csv",header=T)
df <- melt(df, id.vars="Size")
names(df)[2] <- "Type"
names(df)[3] <- "Time"

summary <- df %>% group_by(Type, Size) %>% summarise_at(vars(Time), list(meanTime = mean))

ggplot(df, aes(colour=Type)) +
  geom_point(data = df, aes(x = Size, y = Time), size = .1) +
  geom_point(data = summary, aes(x = Size, y = meanTime), size = 1) +
  geom_line(data = summary, aes(x = Size, y = meanTime), size = .7) +
  labs(title = "\"Time\" spent by Machine 1 to sort an array of a given \"Size\"") +
  theme( plot.title = element_text(hjust = 0.5)) + theme_light() +
  scale_x_continuous(breaks=c(0, 8000000, 12000000, 15000000))
```



Comment: For arrays with a size of up to 11000000 elements, the built-in method is the most rapid. Afterwards, the Threaded computing performs better. Also, for small array sizes (less than 8000000) the use of Parallel computing only slows down the speed, and we see that the Sequential sorting provides quicker results.

One clear reason why the parallel method was slow with smaller array size refer to the fact that the joining process of the thread sides take a considerable amount of time compared to the whole latency of the sequential and the build-in methods.

Note: This experiment was done on a virtual machine (4 cores assigned, 8 GB of RAM), and there was no other application running when the experiment was running.

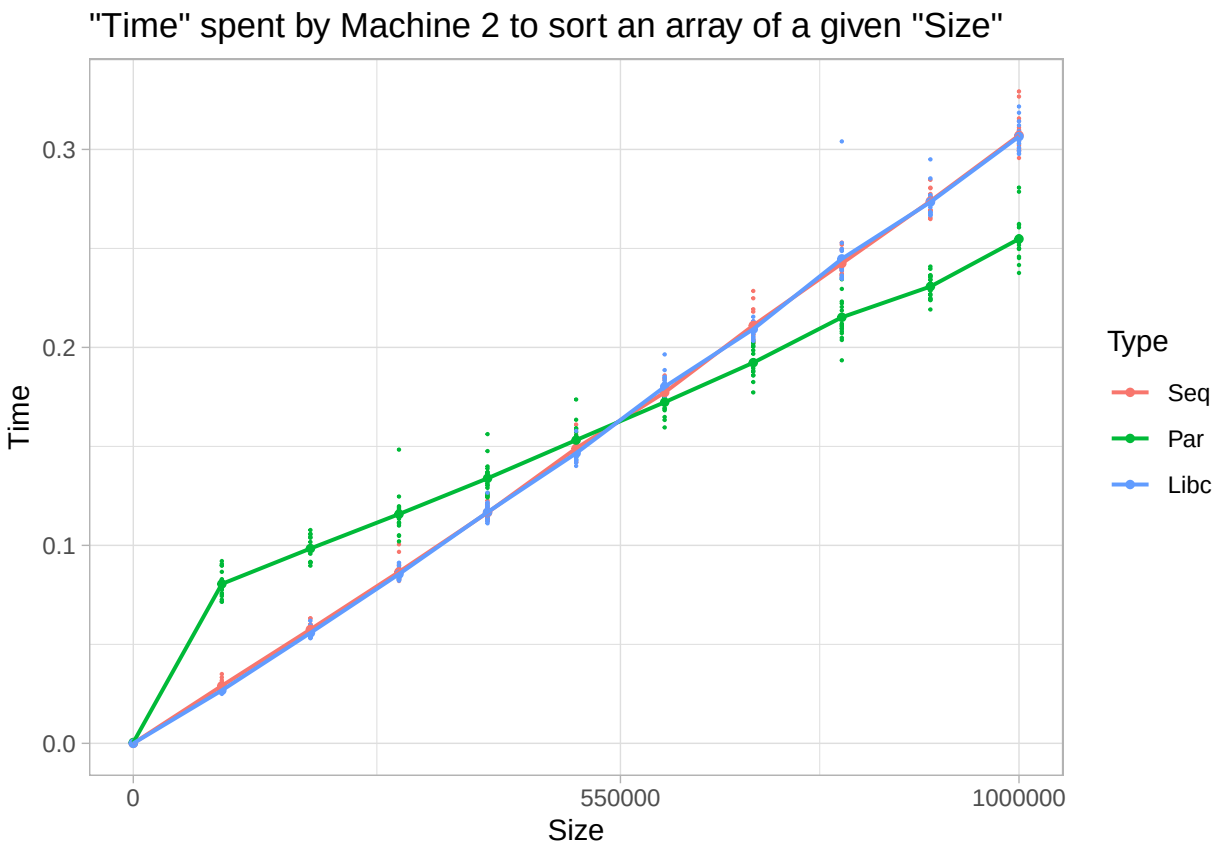
3.2. Machine 2 (Physical Machine): 4 cores, 2 threads per core, 1346 CPU MHz

Resulting data: `data/oulkaido_2021-12-08/measurements_20:08.csv`

```
#parameters: START_SIZE=0, MAX_SIZE=1000000, STEP=100000
df2 <- read.csv("data/oulkaido_2021-12-08/measurements_20:08.csv",header=T)
df2 <- melt(df2, id.vars="Size")
names(df2)[2] <- "Type"
names(df2)[3] <- "Time"

summary2 <- df2 %>% group_by(Type, Size) %>% summarise_at(vars(Time), list(meanTime = mean))

ggplot(df2, aes(colour=Type)) +
  geom_point(data = df2, aes(x = Size, y = Time), size = .1) +
  geom_point(data = summary2, aes(x = Size, y = meanTime), size = 1) +
  geom_line(data = summary2, aes(x = Size, y = meanTime), size = .7) +
  labs(title = "\"Time\" spent by Machine 2 to sort an array of a given \"Size\"") +
  theme( plot.title = element_text(hjust = 0.5)) + theme_light() +
  scale_x_continuous(breaks=c(0, 550000, 1000000))
```



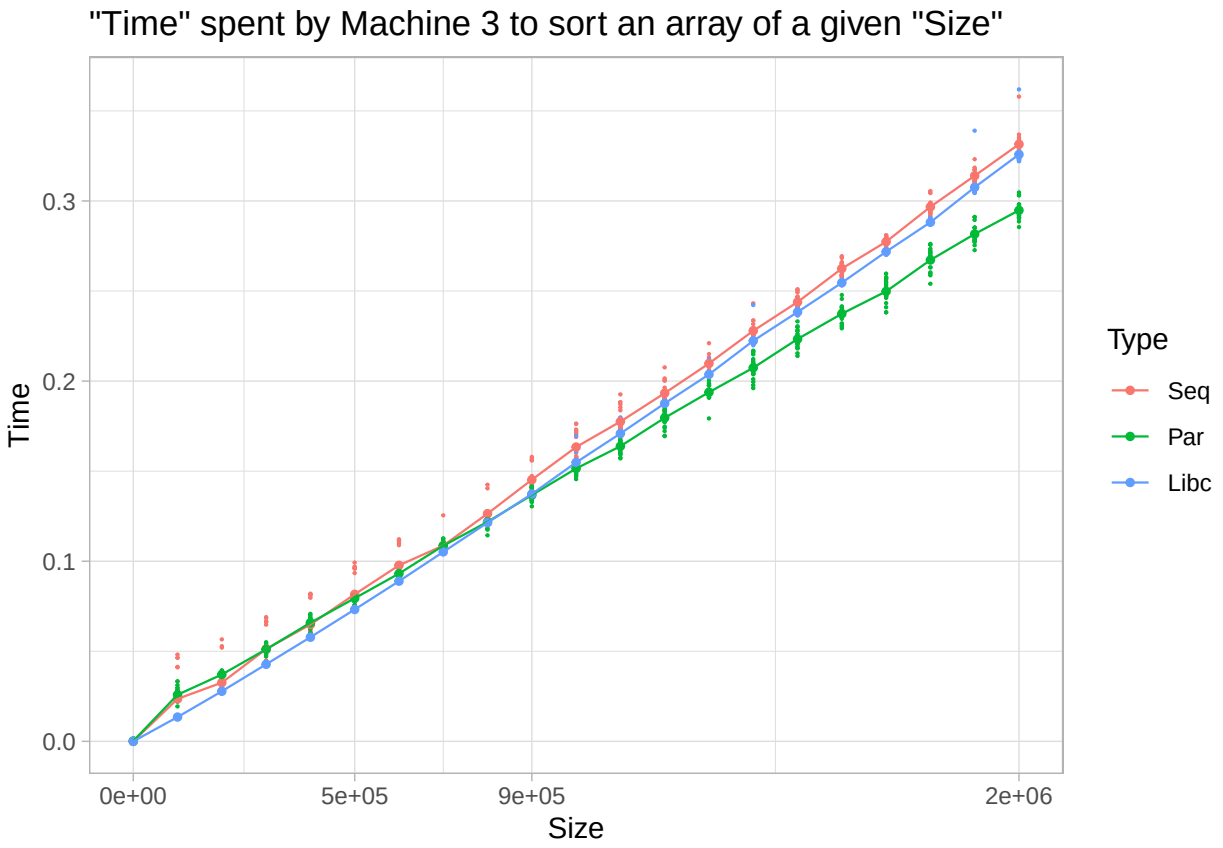
3.3. Machine 3 (Physical Machine): 6 cores, 1 thread per core, 3000 CPU MHz

Resulting data: `data/ensipc203_2021-12-08/measurements_21:39.csv`

```
#parameters: START_SIZE=0, MAX_SIZE=10000000, STEP=500000
df3 <- read.csv("data/ensipc203_2021-12-08/measurements_21:39.csv",header=T)
df3 <- melt(df3, id.vars="Size")
names(df3)[2] <- "Type"
names(df3)[3] <- "Time"

summary3 <- df3 %>% group_by(Type, Size) %>% summarise_at(vars(Time), list(meanTime = mean))

ggplot(df3, aes(colour=Type)) +
  geom_point(data = df3, aes(x = Size, y = Time), size = .1) +
  geom_point(data = summary3, aes(x = Size, y = meanTime), size = 1) +
  geom_line(data = summary3, aes(x = Size, y = meanTime), size = .4) +
  labs(title = "\"Time\" spent by Machine 3 to sort an array of a given \"Size\"") +
  theme( plot.title = element_text(hjust = 0.5)) + theme_light() +
  scale_x_continuous(breaks=c(0, 500000, 900000, 2000000))
```



3.4. Comparison according to Parallel time

Summary of CPU specification:

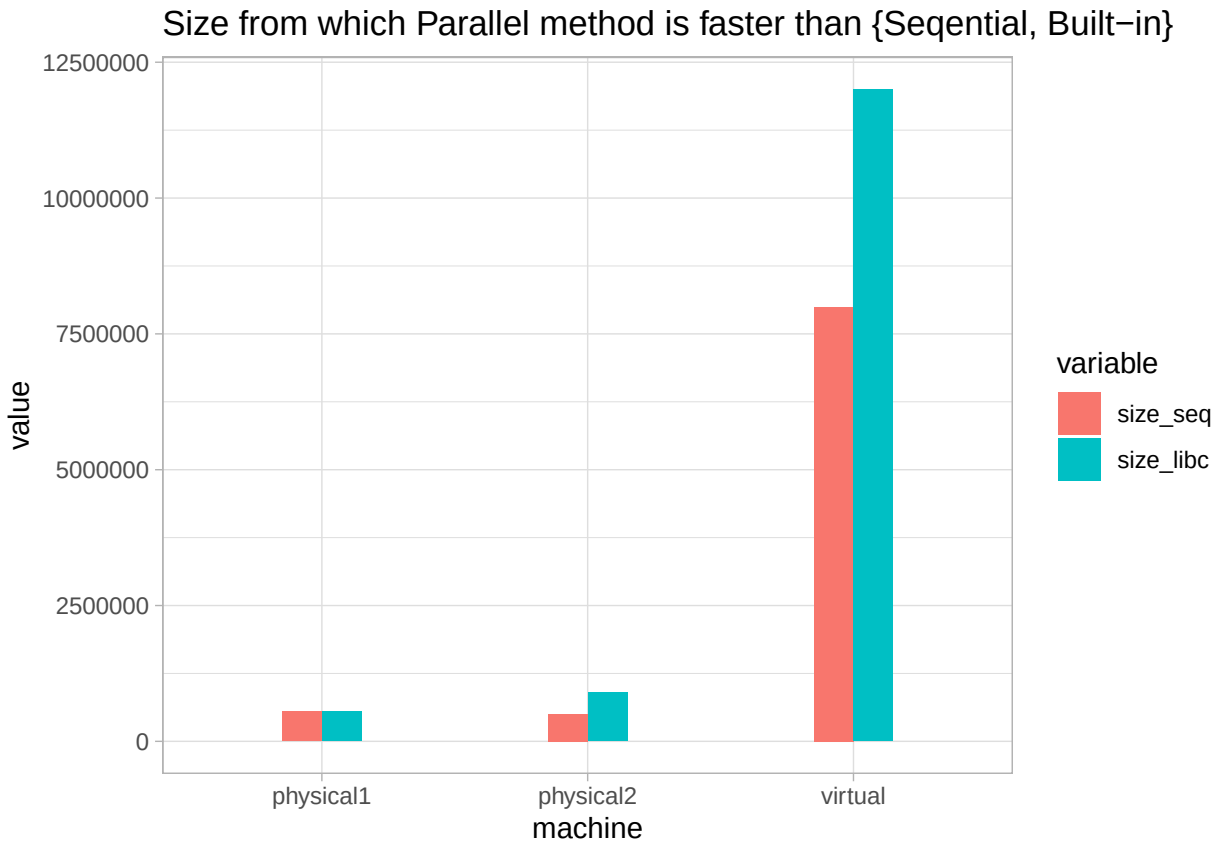
`size_seq` denotes the size from which parallel method become faster than the sequential one. `size_libc` denotes the size from which parallel method become faster than the built-in method.

```
##
## | Machine       | Cores  | Thread/core  | CPU MHz | size_seq | size_libc |
## |---------------|--------|--------------|---------|----------|-----------|
## | 1 (Virtual)   | 4      | 1            | 2370    | 8000000  | 12000000  |
## | 2 (Physical)  | 4      | 2            | 1346    | 550000   | 550000    |
## | 3 (Physcial)  | 6      | 1            | 3000    | 500000   | 900000    |
```

Plot & Comparison:

```
library(reshape2)
df5 <- read.csv("data/comparison.csv",header=T)
mdf <- melt(df5[,c('machine','size_seq','size_libc')], id.vars = 1)

ggplot(mdf, aes(x = machine, y=value)) +
  geom_bar(aes(fill=variable), stat="identity", position = "dodge", width=0.3) +
  labs(title = "Size from which Parallel method is faster than {Seqential, Built-in}") +
  theme( plot.title = element_text(hjust = 0.5)) + theme_light()
```


Size from which Parallel method is faster than {Seqential, Built−in}
12500000
10000000
7500000
5000000
2500000
0
value
physical1
physical2
virtual
machine
variable
size_seq
size_libc