Introduction

In this report, we compare the performance, in term of *run time*, of the *parallel quicksort*, as implemented by **Arnaud Legrand** (https://github.com/alegrand/M2R-ParallelQuicksort)

Experiment machine

The tests where conduced on a 2011 MacBook Pro, with an `Intel(R) Core(TM) i7-2635QM CPU @ 2.00GHz` 4 core processor (with intel's hyperthreading technologies this means 8 virtual cores), 8 Gigabytes of ram, and running Fedora Linux 35. We tried to stop all programs while running the experiment.

Fixing problems with the original experiment

Increasing the robustness of the testing script

A test script is provided by default on this project :

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

```
OUTPUT_DIRECTORY=data/`hostname`_`date +%F`
mkdir -p $OUTPUT_DIRECTORY
OUTPUT_FILE=$OUTPUT_DIRECTORY/measurements_`date +%R`.txt

touch $OUTPUT_FILE
for i in $(seq 1000000 1000000 10000000) ; do
    for rep in `seq 1 5`; do
        echo "Size: $i" >> $OUTPUT_FILE;
        ./src/parallelQuicksort $i >> $OUTPUT_FILE;
    done ;
done
```

While this script is fine, it can be enhanced to be made more robust. As suggested by **Arnaud Legrand** the main modification will be the interleaving of the execution. Our script now runs : - small medium large small medium large small medium large
instead of the original - small small small medium medium medium large large large

Additionally, we will collect data in static increment instead of powers of 10 as done originally, to have more data on all size :

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

```
OUTPUT_DIRECTORY=data/`hostname`_`date +%F`
mkdir -p $OUTPUT_DIRECTORY
OUTPUT_FILE=$OUTPUT_DIRECTORY/measurements_`date +%R`.txt

touch $OUTPUT_FILE
for i in $(seq 1 5); do
    for j in $(seq 0 1000000 10000000) ; do
        echo "Size: $j" >> $OUTPUT_FILE;
        ./src/parallelQuicksort $j >> $OUTPUT_FILE;

    done
done
```

Modifying the `MakeFile` for optimisation

As pointed out by the professor **Arnaud Legrand** again the executable is compiled with `-O0`. For a performance test, it seems more relevant to use `-O3`, even if `-O0` is the default. Additionally, we also found that the executable is also compiled with all sort of strange options, that we don't think are necessary as well. Most notably, `-g` is turned on, which is the option for debugging, however for testing the performance it seems more relevant to disable this option. Bellow you can compare our new Makefile vs the oldMakefile.

```
echo "NEW MAKEFILE"
cat src/Makefile
echo
echo
echo "OLD MAKEFILE"
cat src/oldMakefile
```

```
NEW MAKEFILE
parallelQuicksort: parallelQuicksort.o




CFLAGS = -Wall -O3 -pthread -lrt -std=c99

%: %.o
    $(CC) $(INCLUDES) $(DEFS) $(CFLAGS) $^ $(LIBS) -o $@

%.o: %.c
    $(CC) $(INCLUDES) $(DEFS) $(CFLAGS) -c -o $@ $<

clean:
    rm -f parallelQuicksort *.o *~


OLD MAKEFILE
parallelQuicksort: parallelQuicksort.o

PEDANTIC_PARANOID_FREAK =       -g -Wall -Wshadow -Wcast-align \
                -Waggregate-return -Wmissing-prototypes -Wmissing-declarations \
                -Wstrict-prototypes -Wmissing-prototypes -Wmissing-declarations \
                -Wmissing-noreturn  \
                -Wpointer-arith -Wwrite-strings -finline-functions -O3

REASONABLY_CAREFUL_DUDE =   -g -Wall -O2
NO_PRAYER_FOR_THE_WICKED =  -w -O3
WARNINGS =          $(PEDANTIC_PARANOID_FREAK)
CFLAGS = $(WARNINGS) -pthread -lrt -std=c99
LIBS =

%: %.o
    $(CC) $(INCLUDES) $(DEFS) $(CFLAGS) $^ $(LIBS) -o $@

%.o: %.c
    $(CC) $(INCLUDES) $(DEFS) $(CFLAGS) -c -o $@ $<

clean:
    rm -f parallelQuicksort *.o *~
```

Results

We have 2 objectives in this experiment:

- Finding if the parallel version is faster
- If yes, finding at what input size it is faster, because of course, the overhead of parallelisation is never worth the cost on very small inputs, so there will be a crosspoint when parallel becomes faster.

We ran one from size 0 to 10^7 by increment of 10^6. We used the perl script, `scripts/csv_quicksort_extractor.pls`, to transform the output to csv in order to be able to import it in `R`.

```
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)
require(scales)
```

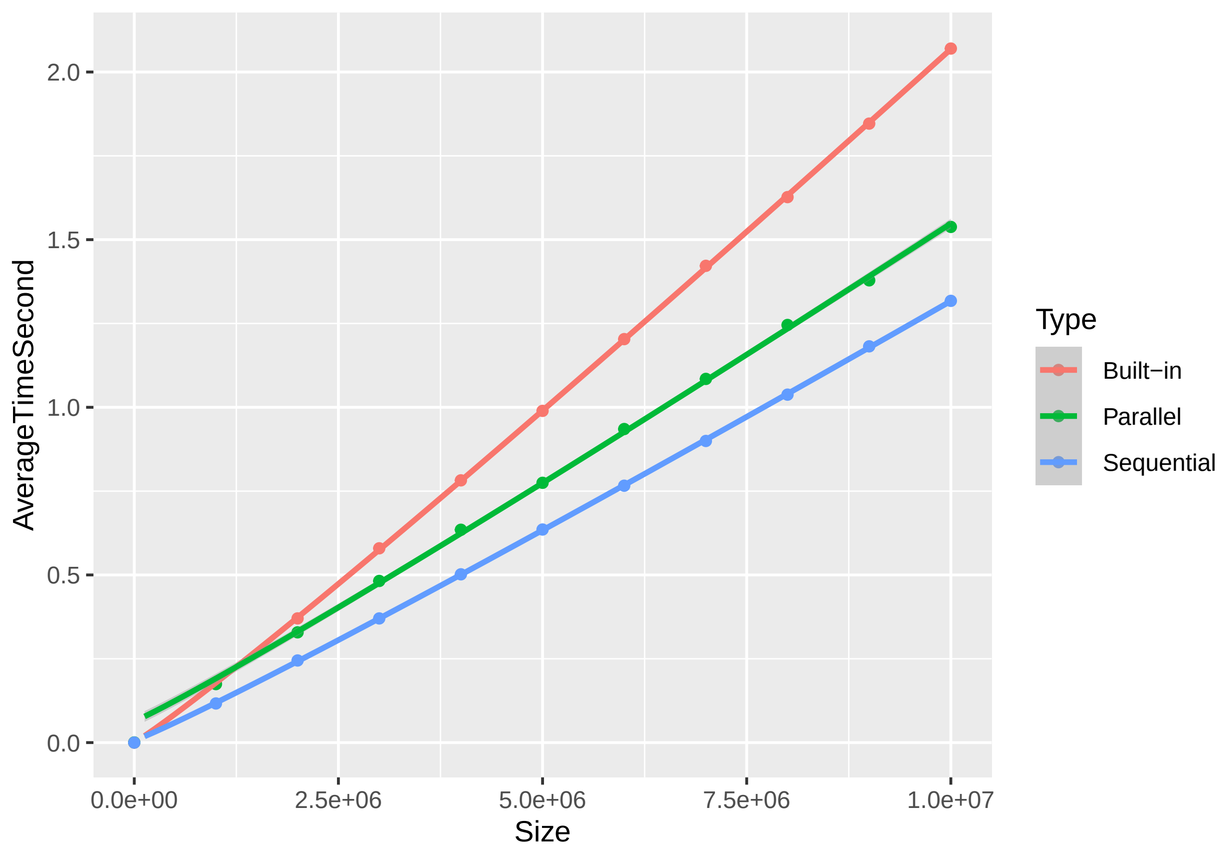
```
## Le chargement a nécessité le package : scales
```

```
library(vroom)


df = read.csv("data/fedora_2021-11-21/measurements_11:13.csv") # load the data from the CSV


df = df %>% group_by(Type, Size) %>% summarise(AverageTimeSecond=mean(Time)) # group by type and size a
```

```
## `summarise()` has grouped output by 'Type'. You can override using the `.groups` argument.
```

```
p <- ggplot(df, aes(x=Size, y=AverageTimeSecond, color=Type))
p <- p  + geom_point()  + geom_smooth(method = 'lm',formula = y ~I(x*log(x)))# we use geom smooth to ge

print(p)
```

```
## Warning: Removed 3 rows containing missing values (geom_smooth).
```



The weird thing around 0 is because the point for all 3 algorithms are all in the same place.

Conclusion

The parallel version becomes significantly faster than the built-in quicksort at around size 10^6. However it seems that it's slower than the custom sequential version at any size, and it gets slower and slower as the size gets larger.