Parallel Quicksort Experiments

Yacine Ndiaye

4 Mars 2016

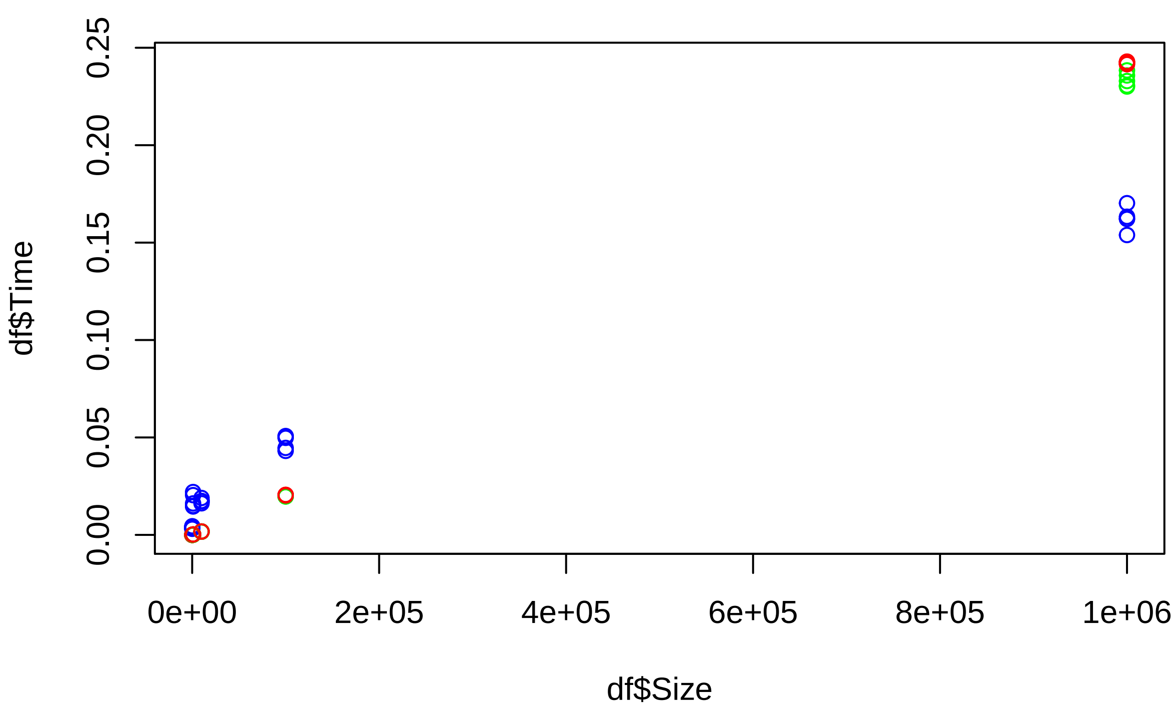
___Experiments in class

We did these tests in class to learn some experimental and plotting methods

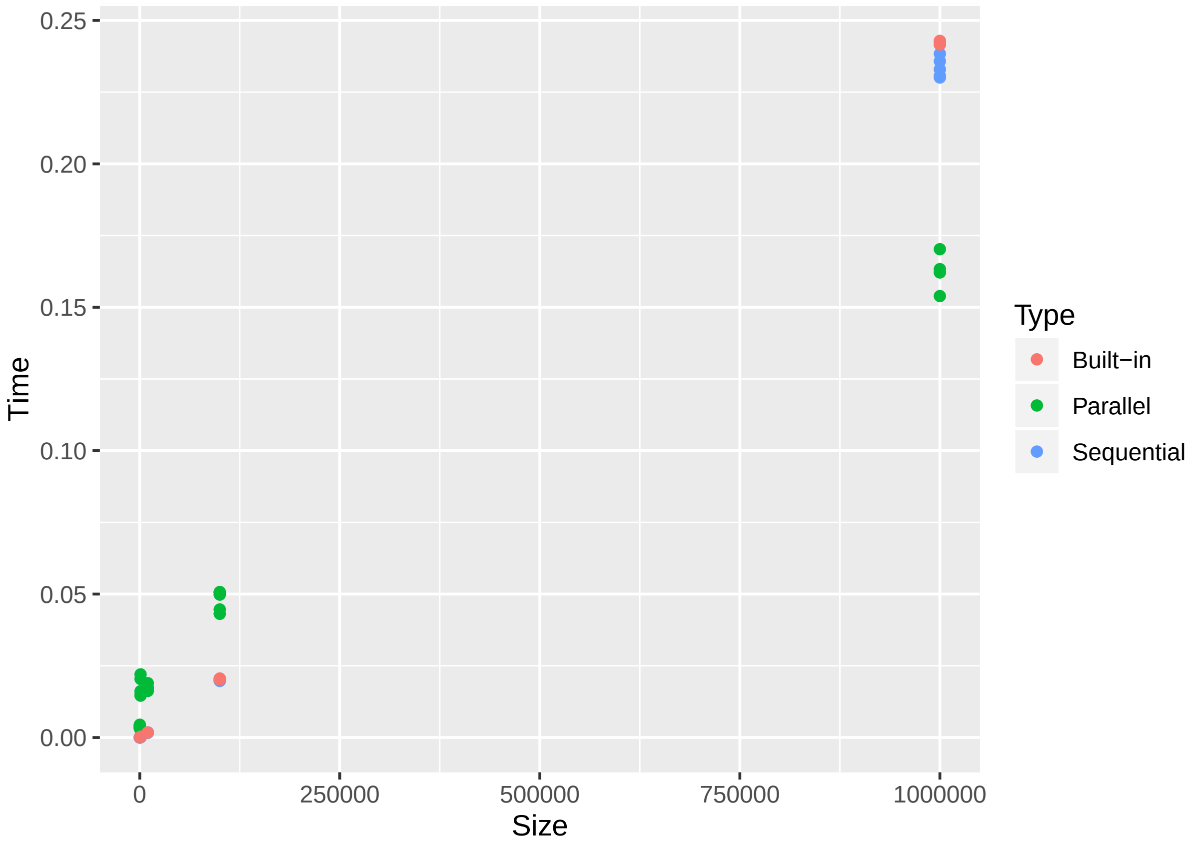
```
#install.packages("ggplot2")
set.seed(42)
library(ggplot2)
library(plyr)

  df <- read.csv("/home/yacine/Documents/performance/M2R-ParallelQuicksort/data/sama_2014-10-13/measure
  head(df)
```

```
##   Size       Type     Time
## 1  100 Sequential 0.000010
## 2  100   Parallel 0.004024
## 3  100   Built-in 0.000013
## 4  100 Sequential 0.000010
## 5  100   Parallel 0.004448
## 6  100   Built-in 0.000014
```

```
  plot(df$Size,df$Time,col=c("red","blue","green")[df$Type])
```



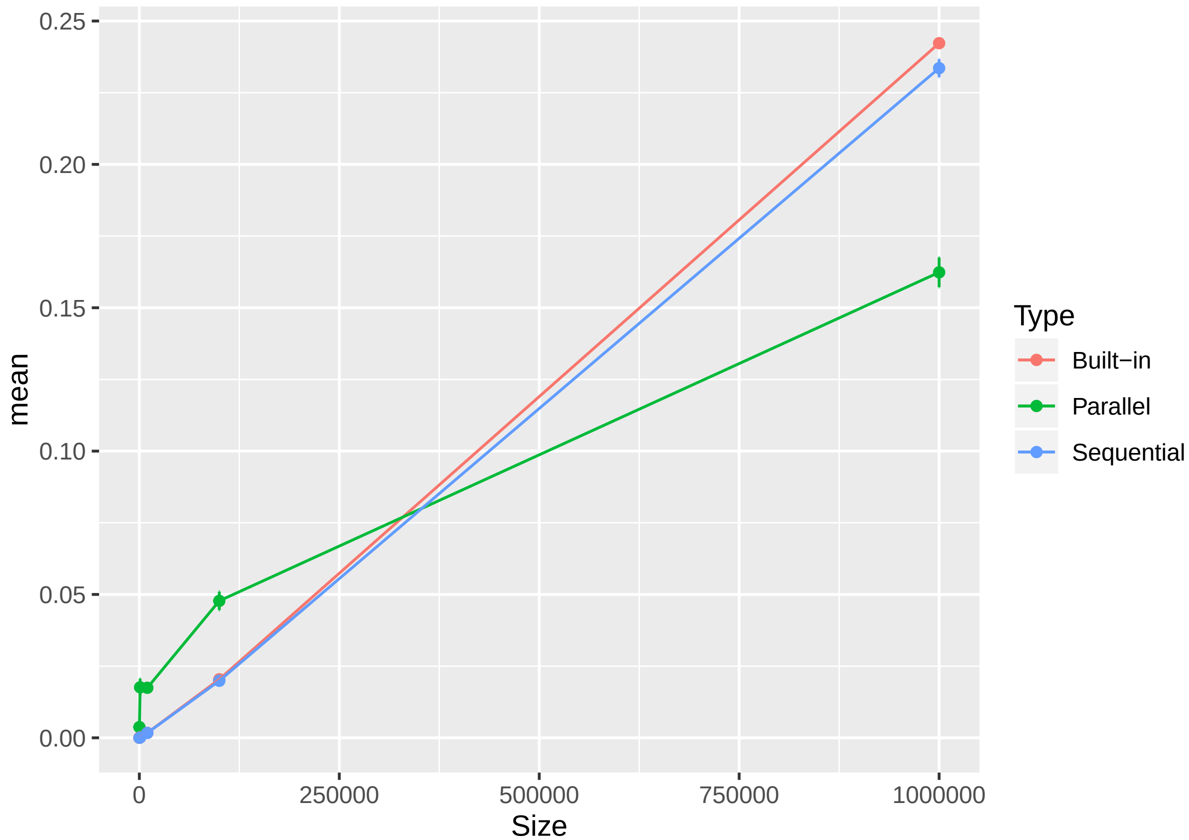
```r
ggplot(data=df, aes(x=Size, y=Time, color=Type))+geom_point()
```




```r
df_sum = ddply(df, c("Size", "Type"), summarize, num=length(Time), mean=mean(Time), sd= sd(Time), se=s
df_sum
```

```
##       Size       Type num      mean           sd           se
## 1      100   Built-in   5 0.0000126 1.140175e-06 1.019804e-06
## 2      100   Parallel   5 0.0037454 5.188842e-04 4.641041e-04
## 3      100 Sequential   5 0.0000098 4.472136e-07 4.000000e-07
## 4     1000   Built-in   5 0.0002078 3.834058e-06 3.429286e-06
## 5     1000   Parallel   5 0.0176116 3.378959e-03 3.022233e-03
## 6     1000 Sequential   5 0.0001278 1.095445e-06 9.797959e-07
## 7    10000   Built-in   5 0.0017194 1.165333e-05 1.042305e-05
## 8    10000   Parallel   5 0.0174410 9.699515e-04 8.675510e-04
## 9    10000 Sequential   5 0.0016958 4.669261e-05 4.176314e-05
## 10  100000   Built-in   5 0.0204072 1.263555e-04 1.130158e-04
## 11  100000   Parallel   5 0.0477688 3.609278e-03 3.228237e-03
## 12  100000 Sequential   5 0.0198892 1.405763e-04 1.257353e-04
## 13 1000000   Built-in   5 0.2422674 6.296517e-04 5.631776e-04
## 14 1000000   Parallel   5 0.1623540 5.800859e-03 5.188446e-03
## 15 1000000 Sequential   5 0.2335652 3.502431e-03 3.132669e-03
```

```r
ggplot(data=df_sum,aes(x=Size, y=mean, ymin=mean-se, ymax= mean+se, color=Type))+geom_errorbar()+geom_p
```




```
#ggplot(data=df, aes(x=Size, y=Time, color=factor(Type), shape=factor(option_compil)))+geom_point()
```

___First Experiments

___ Experiment 1: The sizes

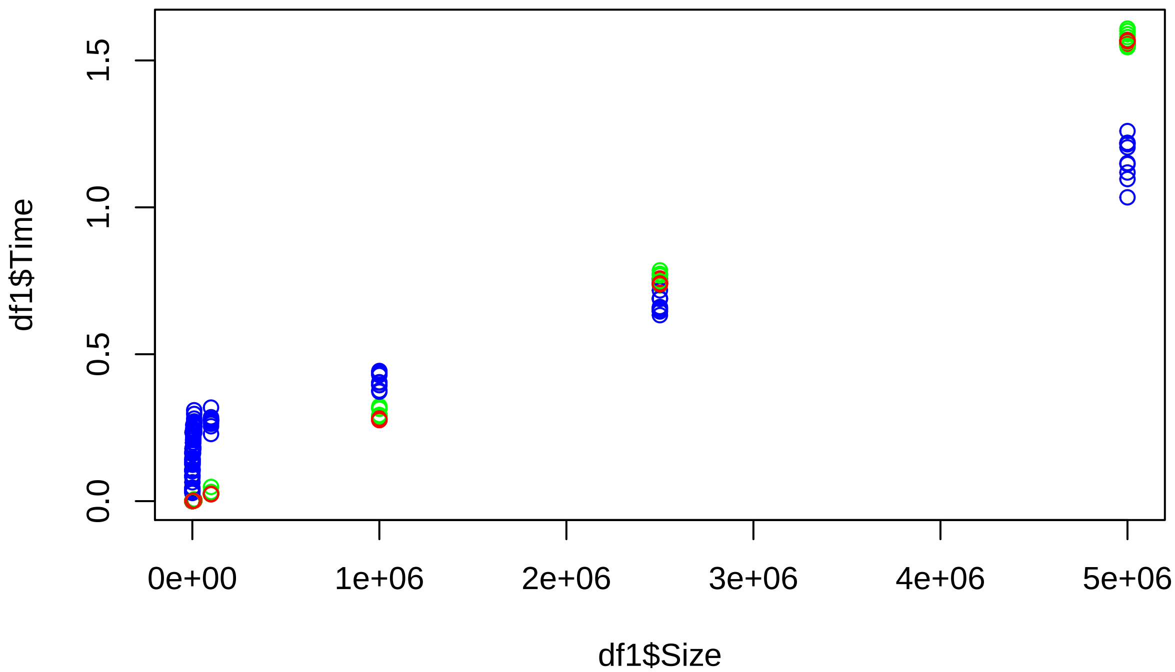
Instead of increasing the size of the array gradually, let's try to choose different sizes in a pretty mixed way. In the following line, we have the array sizes we use in the script:

1000 2500000 10000 100 5000 1000000 800 100000 430 5000000 4000

```
library(ggplot2)
library(plyr)

df1 <- read.csv("/home/yacine/Documents/performance/M2R-ParallelQuicksort/data/yacine-S550CA_2016-01-3
head(df1)
```

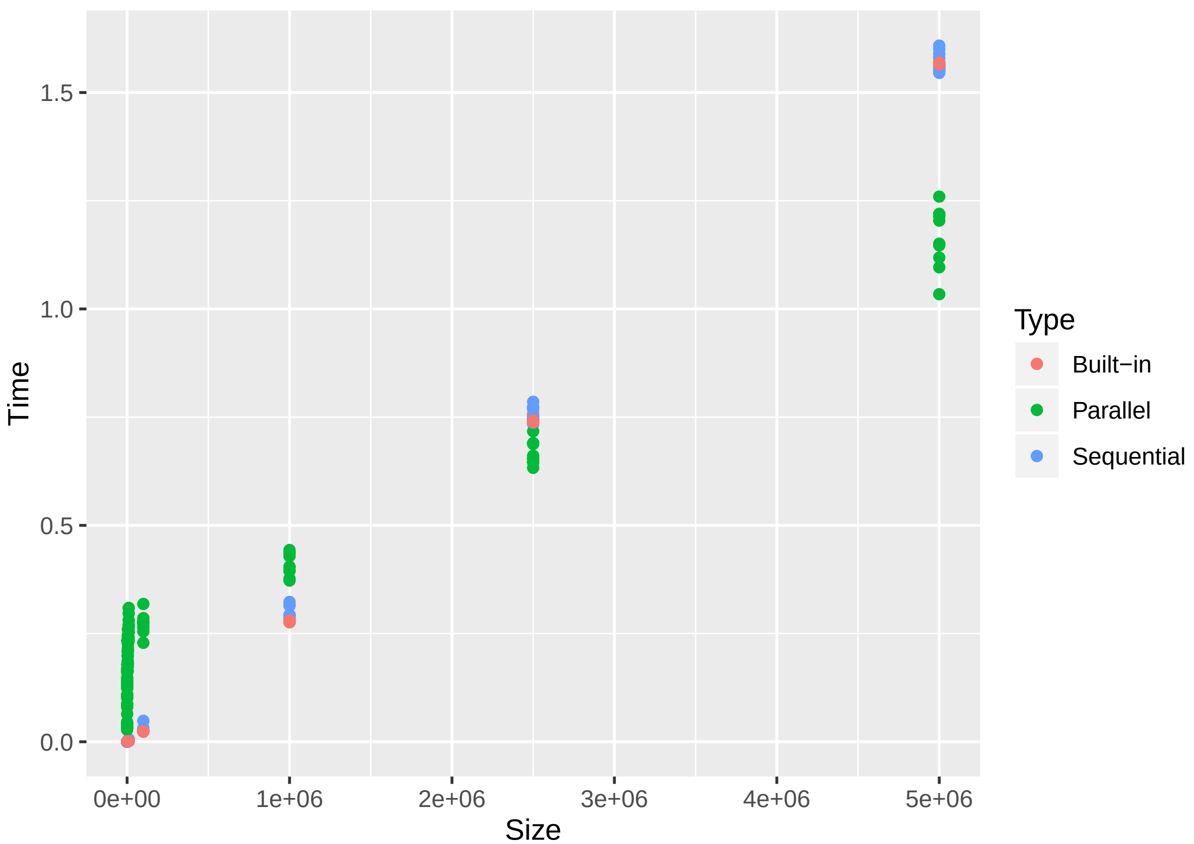
```
##   Size       Type     Time
## 1 1000 Sequential 0.000383
## 2 1000   Parallel 0.140358
## 3 1000   Built-in 0.000257
## 4 1000 Sequential 0.000164
## 5 1000   Parallel 0.108913
## 6 1000   Built-in 0.000257
```

```r
plot(df1$Size,df1$Time,col=c("red","blue","green")[df1$Type])
```



Let's see the different execution times with ggplot

```r
ggplot(data=df1, aes(x=Size, y=Time, color=Type))+geom_point()
```



We can clearly see that the parallel quick sort is not very efficient for little array sizes. The built in quick sort is the best, then the sequential and the parallel one is the worst when we have little arrays. This tend to change after the 2.5M size. When we have a 5M size, it is considerably better to use the parallel quick sort.

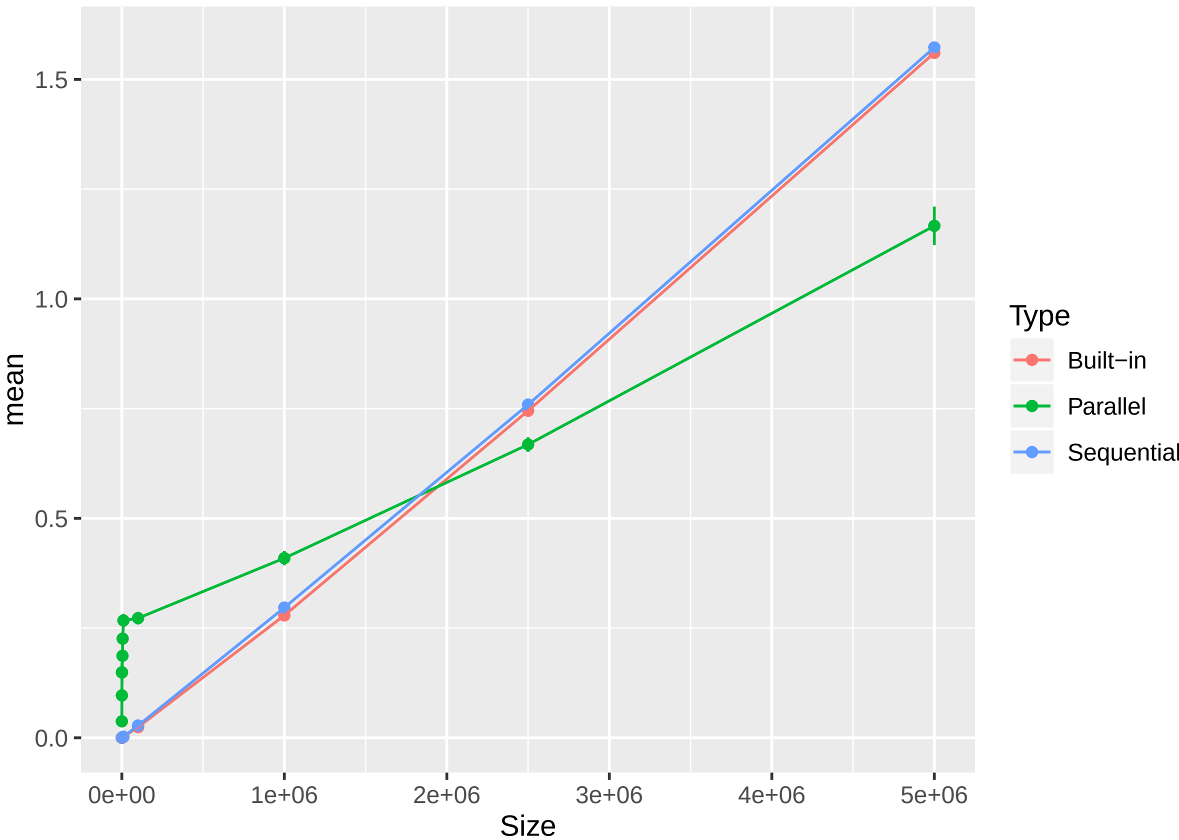
___Confidence interval

Now let's see the confidence interval

```
  df1_sum = ddply(df1, c("Size", "Type"), summarize, num=length(Time), mean=mean(Time), sd= sd(Time),
df1_sum
```

```
##        Size       Type num      mean           sd           se
## 1       100   Built-in  10 0.0000158 4.216370e-07 2.666667e-07
## 2       100   Parallel  10 0.0375457 5.667356e-03 3.584351e-03
## 3       100 Sequential  10 0.0000132 1.135292e-06 7.180220e-07
## 4       430   Built-in  10 0.0001748 1.619328e-06 1.024153e-06
## 5       430   Parallel  10 0.0965844 1.728921e-02 1.093466e-02
## 6       430 Sequential  10 0.0000677 5.478240e-06 3.464743e-06
## 7       800   Built-in  10 0.0002386 5.521674e-06 3.492214e-06
## 8       800   Parallel  10 0.1490409 1.709129e-02 1.080948e-02
## 9       800 Sequential  10 0.0001589 8.798163e-05 5.564447e-05
## 10     1000   Built-in  10 0.0002564 4.427189e-06 2.800000e-06
## 11     1000   Parallel  10 0.1489734 3.476853e-02 2.198955e-02
## 12     1000 Sequential  10 0.0001853 6.948389e-05 4.394547e-05
## 13     4000   Built-in  10 0.0008529 1.508826e-05 9.542653e-06
## 14     4000   Parallel  10 0.1867944 1.381444e-02 8.737016e-03
## 15     4000 Sequential  10 0.0007487 3.019952e-05 1.909985e-05
## 16     5000   Built-in  10 0.0010449 1.760335e-05 1.113333e-05
## 17     5000   Parallel  10 0.2259011 2.244976e-02 1.419848e-02
## 18     5000 Sequential  10 0.0009795 2.876437e-05 1.819218e-05
## 19    10000   Built-in  10 0.0021134 2.278986e-05 1.441357e-05
## 20    10000   Parallel  10 0.2672344 2.375610e-02 1.502467e-02
## 21    10000 Sequential  10 0.0025239 1.284060e-03 8.121109e-04
## 22   100000   Built-in  10 0.0243268 5.792073e-04 3.663228e-04
## 23   100000   Parallel  10 0.2725303 2.287437e-02 1.446702e-02
## 24   100000 Sequential  10 0.0279665 7.560262e-03 4.781529e-03
## 25  1000000   Built-in  10 0.2787734 2.660901e-03 1.682902e-03
## 26  1000000   Parallel  10 0.4092811 2.570134e-02 1.625496e-02
## 27  1000000 Sequential  10 0.2966316 1.542580e-02 9.756131e-03
## 28  2500000   Built-in  10 0.7450571 6.725889e-03 4.253826e-03
## 29  2500000   Parallel  10 0.6680882 2.657049e-02 1.680465e-02
## 30  2500000 Sequential  10 0.7590628 1.754608e-02 1.109711e-02
## 31  5000000   Built-in  10 1.5603784 6.502567e-03 4.112585e-03
## 32  5000000   Parallel  10 1.1663412 6.940289e-02 4.389424e-02
## 33  5000000 Sequential  10 1.5728140 2.112513e-02 1.336071e-02
```

```
ggplot(data=df1_sum,aes(x=Size, y=mean, ymin=mean-se, ymax= mean+se, color=Type))+geom_errorbar()+geom
```



Here we can see after applying the confidence interval that there is the very slight difference between the built in and the sequential algorithms for a little array size but when the size is increasing, the built in algorithm is a little bit better. For the parallel quick sort,like we said before, it is only efficient starting a certain array size and then we can notice the exact opposite of time evolution compared to the other 2 algorithms.

___Experiment 2 : The GCC compiler options

```
df2 <- read.csv("/home/yacine/Documents/performance/M2R-ParallelQuicksort/data/yacine-S550CA_2016-02-0
 head(df2)
```

```
##   Size Compilation       Type     Time
## 1 1000         -O1  Sequential 0.000290
## 2 1000         -O1    Parallel 0.172405
## 3 1000         -O1    Built-in 0.000251
## 4 1000         -O2  Sequential 0.000082
## 5 1000         -O2    Parallel 0.108279
## 6 1000         -O2    Built-in 0.000247
```

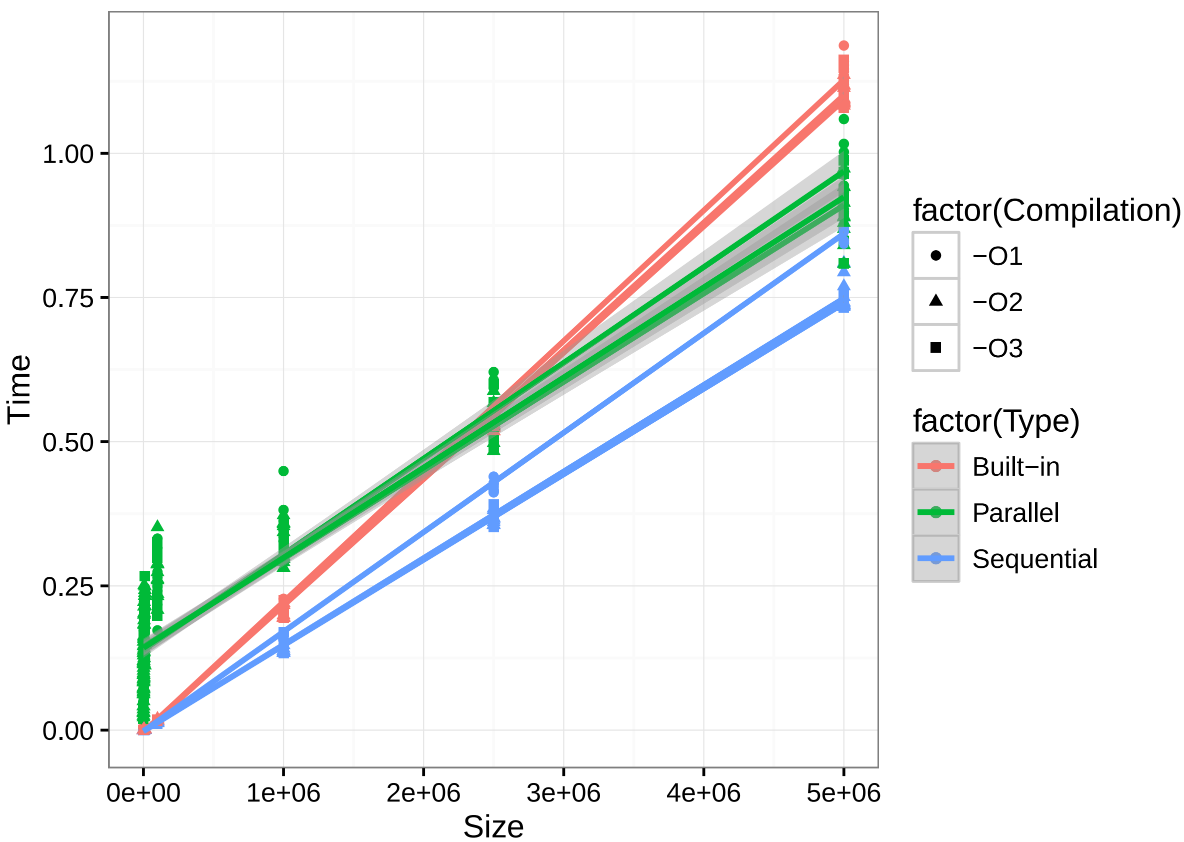
Let's see the different execution times with ggplot

```
 library (dplyr)
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:plyr':
##
##     arrange, count, desc, failwith, id, mutate, rename, summarise,
##     summarize

## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
graph=ggplot(data=df2, aes(x=Size, y=Time, color=factor(Type), shape= factor(Compilation)) )+geom_point
graph
```

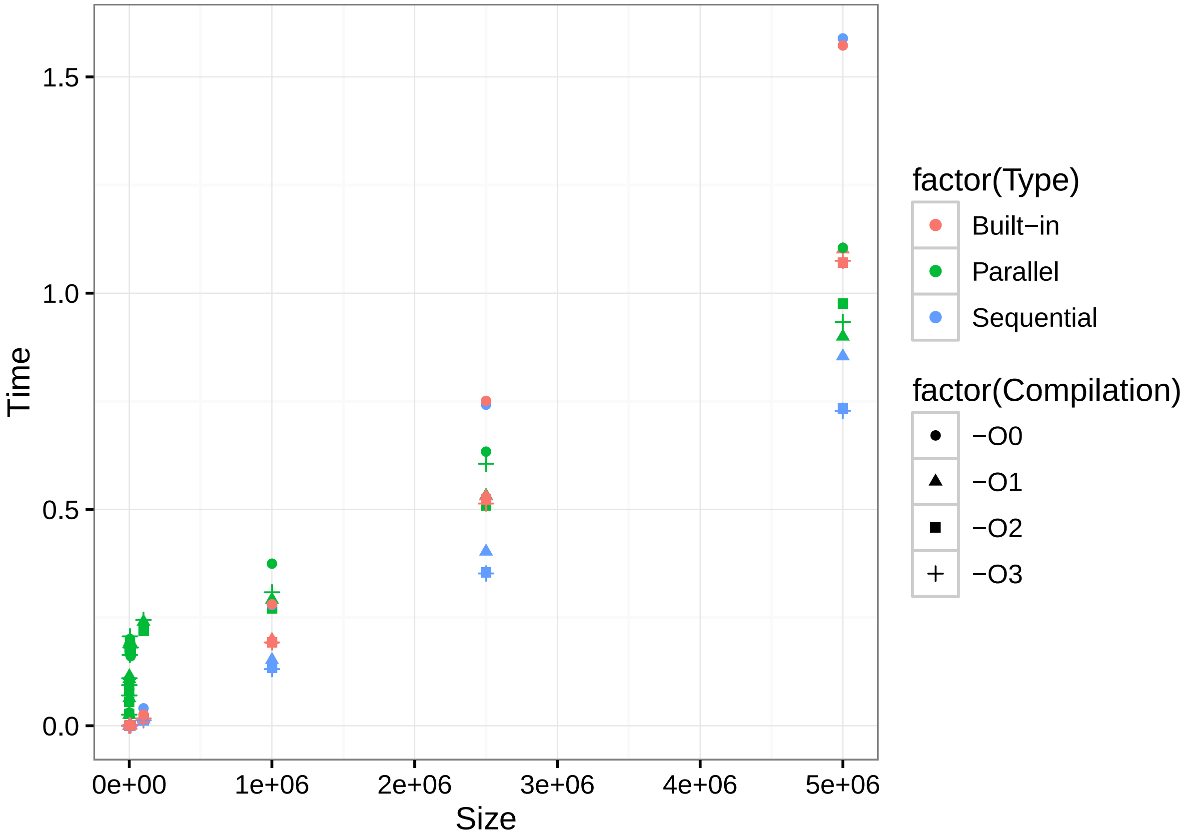



```
#graph + coord_cartesian(xlim = c(0,1000000))
```

The plot show us that when using optimization compiling options, the sequential quick sort can be very efficient (with my machine). That's weird because without the gcc optimization options, we did had before better results with the parallel quick sort. So what happened? Maybe it's because of the -00 option that i didn't take into account (We had it before in the makefile but i removed it one moment to debug). So, let's add the -O0 option in the script.

It is not really easy to follow with that much information, so now we are just going to do the experiment 1 time for a specific array size.

```r
df3 <- read.csv("/home/yacine/Documents/performance/M2R-ParallelQuicksort/data/yacine-S550CA_2016-03-03/

ggplot(data=df3, aes(x=Size, y=Time, color=factor(Type), shape= factor(Compilation)) )+geom_point()+ther
```



It's a little bit better, as we can see with the graph above, the parallel quick sort is now better for a big array size,as we had before, but only for a certain compiling option (now we know that it is the -0O option), a result we can also notice around the 5e+06 sizes (the -O0 is represented by the dots).

To conclude, we can say that we must use the -O0 option if we want to have efficiency with this parallel quick sort. However, the sequential algorithm, with the -O2 option can also be very efficient.

These differences are not very surprising knowing that using the gcc optimization options, the resulting improvement in execution time, both depend on the particular application and the hardware environment (see at the beginning the information about my machine)

Usually it's better to do some experiment to find the best level for the application and that's what we did here. It was a very interesting experiment since the graph can help us choose the right optimization option depending on the size of the array and the quick sort version.

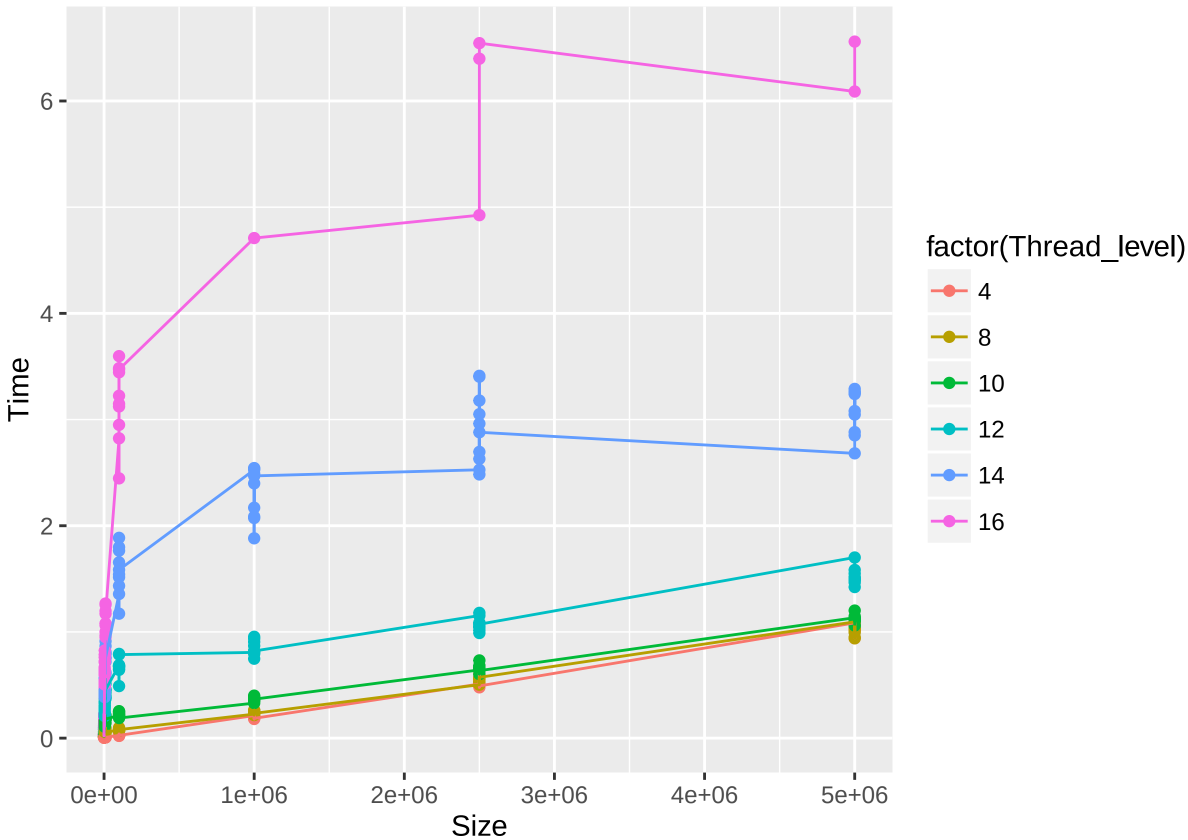
Experiment 3: Thread levels

In this part, we are going to change the thread level in the program. We are going to do some experiments as we did with the array sizes, therefore i will change the code a little bit and as a consequence, create new scripts files to manage the thread levels.

```r
df4 <- read.csv("/home/yacine/Documents/performance/M2R-ParallelQuicksort/data/yacine-S550CA_2016-03-03/
  head(df4)
```

```
##   Size Thread_level      Type     Time
## 1 1000            4  Parallel 0.005228
## 2 1000            4  Parallel 0.007665
## 3 1000            4  Parallel 0.006189
## 4 1000            4  Parallel 0.004442
## 5 1000            4  Parallel 0.004836
## 6 1000            4  Parallel 0.004472
```

```
ggplot(data=df4, aes(x=Size, y=Time, color=factor(Thread_level)) )+geom_point()+geom_line();
```



Now, let's apply the confidence interval

```
 df4_sum = ddply(df4, c( "Size", "Thread_level", "Type"), summarize, num=length(Time), mean=mean(Tim
df4_sum
```

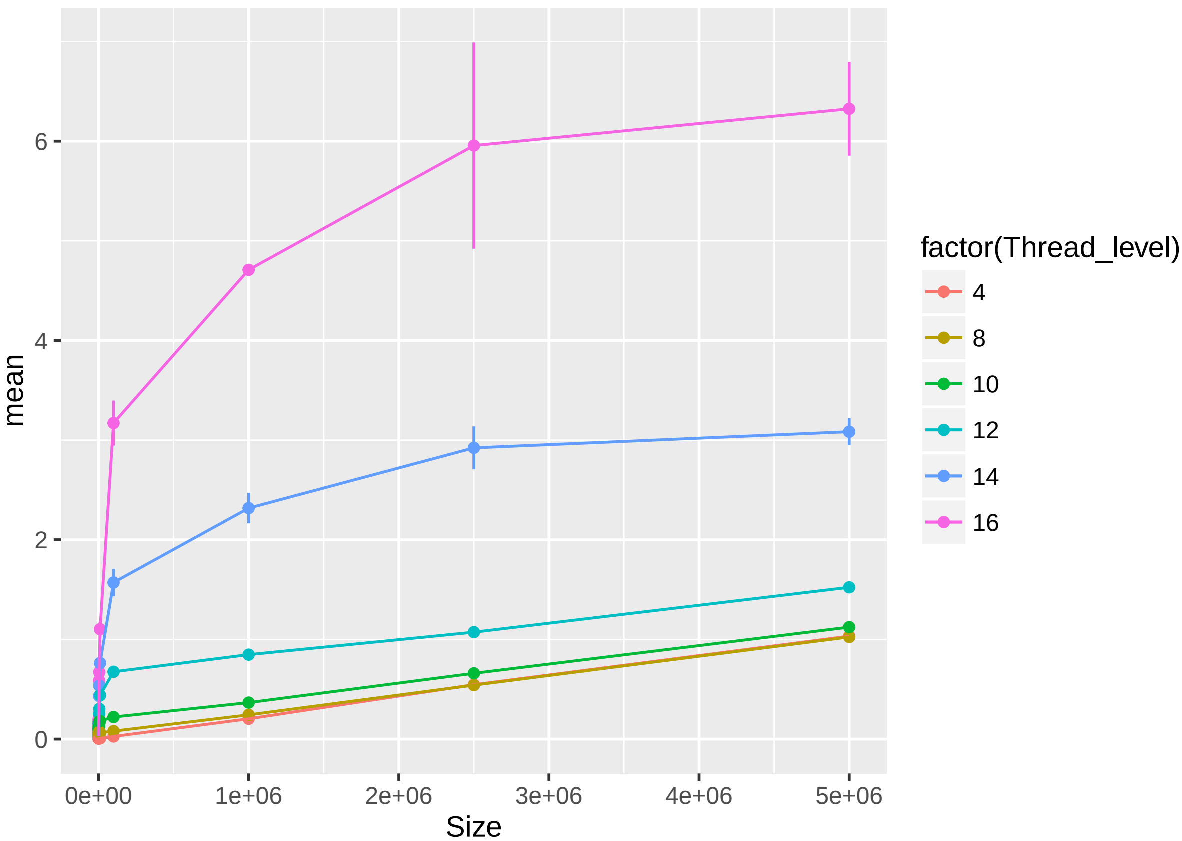
```
##        Size Thread_level      Type num      mean           sd           se
## 1       100            4  Parallel  10 0.0063328 0.0023550452 0.0014894613
## 2       100            8  Parallel  10 0.0198118 0.0043099774 0.0027258690
## 3       100           10  Parallel  10 0.0290712 0.0038646928 0.0024442463
## 4       100           12  Parallel  10 0.0272069 0.0036265978 0.0022936619
## 5       100           14  Parallel  10 0.0306602 0.0048079317 0.0030408030
## 6       100           16  Parallel  10 0.0288859 0.0046290161 0.0029276468
## 7       430            4  Parallel  10 0.0056248 0.0020608485 0.0013033950
## 8       430            8  Parallel  10 0.0349109 0.0047288109 0.0029907626
## 9       430           10  Parallel  10 0.0609036 0.0108171567 0.0068413706
```

```
## 10     430           12  Parallel  10 0.0863674 0.0121352637 0.0076750147
## 11     430           14  Parallel  10 0.0984379 0.0076084338 0.0048119960
## 12     430           16  Parallel  10 0.1205317 0.0100766510 0.0063730337
## 13     800            4  Parallel  10 0.0082623 0.0027668178 0.0017498892
## 14     800            8  Parallel  10 0.0553218 0.0063580192 0.0040211644
## 15     800           10  Parallel  10 0.1086951 0.0143692996 0.0090879430
## 16     800           12  Parallel  10 0.1267349 0.0148559425 0.0093957230
## 17     800           14  Parallel  10 0.1489164 0.0194554610 0.0123047139
## 18     800           16  Parallel  10 0.1685937 0.0150898240 0.0095436427
## 19    1000            4  Parallel  10 0.0054664 0.0010070266 0.0006368995
## 20    1000            8  Parallel  10 0.0385610 0.0044979610 0.0028447603
## 21    1000           10  Parallel  10 0.0981905 0.0190612658 0.0120554030
## 22    1000           12  Parallel  10 0.1540485 0.0362057190 0.0228985073
## 23    1000           14  Parallel  10 0.1782252 0.0248682261 0.0157280471
## 24    1000           16  Parallel  10 0.2008499 0.0215994468 0.0136606896
## 25    4000            4  Parallel  10 0.0076902 0.0024323753 0.0015383692
## 26    4000            8  Parallel  10 0.0686685 0.0091705999 0.0057999967
## 27    4000           10  Parallel  10 0.1337018 0.0182815945 0.0115622956
## 28    4000           12  Parallel  10 0.2539276 0.0417637885 0.0264137390
## 29    4000           14  Parallel  10 0.4324182 0.0366308135 0.0231673607
## 30    4000           16  Parallel  10 0.5867864 0.0572056088 0.0361800037
## 31    5000            4  Parallel  10 0.0081150 0.0006350619 0.0004016484
## 32    5000            8  Parallel  10 0.0756084 0.0062829695 0.0039736988
## 33    5000           10  Parallel  10 0.1775251 0.0188376239 0.0119139594
## 34    5000           12  Parallel  10 0.3027276 0.0765546660 0.0484174220
## 35    5000           14  Parallel  10 0.5363683 0.0591917967 0.0374361792
## 36    5000           16  Parallel  10 0.6721085 0.1220733819 0.0772059857
## 37   10000            4  Parallel  10 0.0074436 0.0007598180 0.0004805511
## 38   10000            8  Parallel  10 0.0625683 0.0128896684 0.0081521421
## 39   10000           10  Parallel  10 0.1816114 0.0187403653 0.0118524477
## 40   10000           12  Parallel  10 0.4395936 0.0601855536 0.0380646863
## 41   10000           14  Parallel  10 0.7634682 0.0970588040 0.0613853775
## 42   10000           16  Parallel  10 1.1028960 0.1154116400 0.0729927301
## 43  100000            4  Parallel  10 0.0264153 0.0050522882 0.0031953476
## 44  100000            8  Parallel  10 0.0801261 0.0097655290 0.0061762629
## 45  100000           10  Parallel  10 0.2216080 0.0217791309 0.0137743318
## 46  100000           12  Parallel  10 0.6755970 0.0832639200 0.0526607268
## 47  100000           14  Parallel  10 1.5710787 0.2169303997 0.1371988313
## 48  100000           16  Parallel  10 3.1712343 0.3567299213 0.2256158122
## 49 1000000            4  Parallel  10 0.2037748 0.0154471561 0.0097696393
## 50 1000000            8  Parallel  10 0.2433212 0.0160381986 0.0101434475
## 51 1000000           10  Parallel  10 0.3661679 0.0205430414 0.0129925602
## 52 1000000           12  Parallel  10 0.8475009 0.0667171701 0.0421956433
## 53 1000000           14  Parallel  10 2.3178775 0.2423729683 0.1532901246
## 54 1000000           16  Parallel   1 4.7090540           NA           NA
## 55 2500000            4  Parallel  10 0.5458965 0.0461280576 0.0291739452
## 56 2500000            8  Parallel  10 0.5413467 0.0243553968 0.0154037054
## 57 2500000           10  Parallel  10 0.6600939 0.0340528040 0.0215368843
## 58 2500000           12  Parallel  10 1.0728945 0.0580663878 0.0367244082
## 59 2500000           14  Parallel  10 2.9222792 0.3404994831 0.2153507818
## 60 2500000           16  Parallel   3 5.9558940 0.8960285565 1.0346446566
## 61 5000000            4  Parallel  10 1.0328871 0.0442821044 0.0280064619
## 62 5000000            8  Parallel  10 1.0246767 0.0497394239 0.0314579738
## 63 5000000           10  Parallel  10 1.1233586 0.0361636305 0.0228718881
```

```
## 64 5000000          12  Parallel  10 1.5228515 0.0764551722 0.0483544966
## 65 5000000          14  Parallel  10 3.0844034 0.2147623711 0.1358276497
## 66 5000000          16  Parallel   2 6.3243670 0.3324038268 0.4700900000
```

```r
ggplot(data=df4_sum,aes(x=Size, y=mean, ymin=mean-se, ymax= mean+se, color=factor(Thread_level)))+geom_
```

```
## Warning: Removed 1 rows containing missing values (geom_errorbar).
```



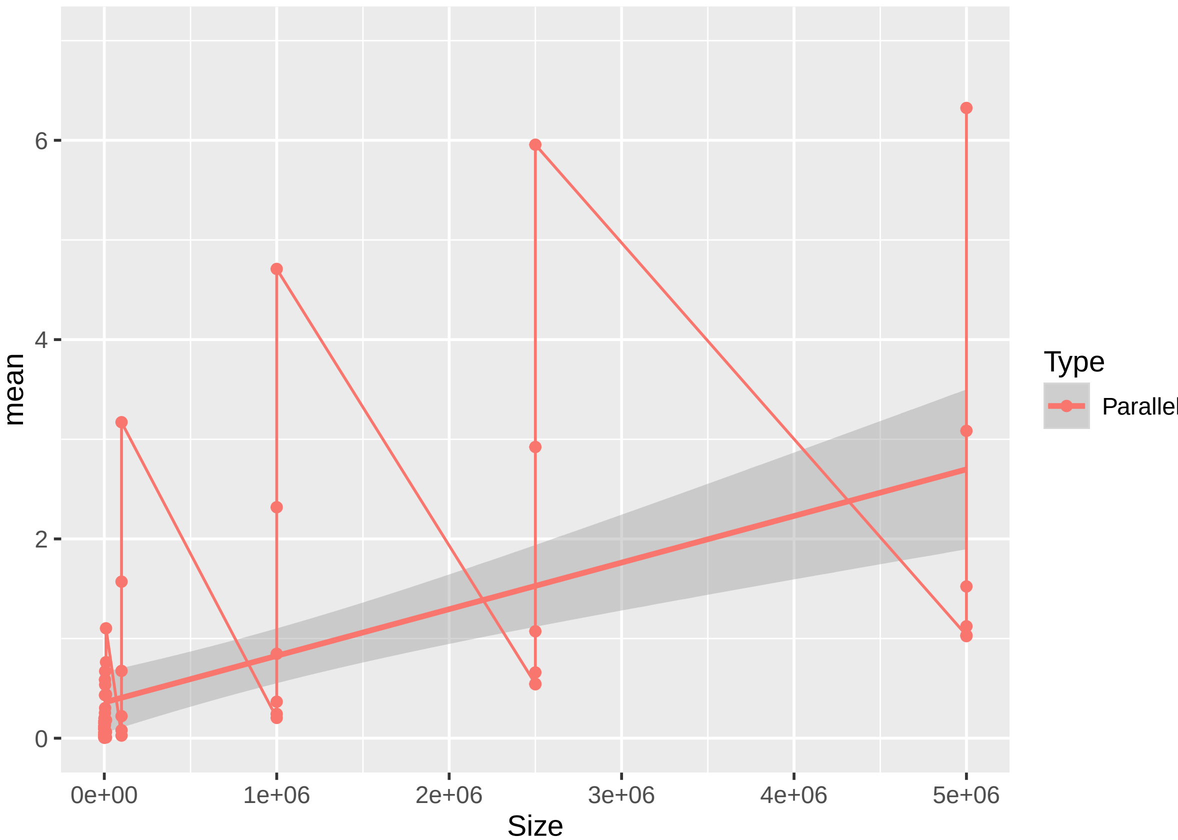
We can see that for little arrays, there is no need to increase the thread level (for my machine). Somehow, using a thread level of 8 is even better than using a thread level of 10. We only consider here the parallel quicksort. Maybe with bigger array sizes, we coulf of see some changes.I can't really say something, it seems like the lower the thread level, the better the execution time.

Experiment 4 :Linear Regression

Now we are going to do some linear regression that can help us make decisions like which array size to use with which algorithm.. The idea is to estimate the values of a and b (Y= a+ bX + epsilon)

Let's see the linear regression of the parallel quick sort since we already have an output for it.

```r
ggplot(data=df4_sum,aes(x=Size, y=mean, ymin=mean-se, ymax= mean+se, color=Type))+geom_smooth(method="]
```




```
#df=data.frame(x=Size,y=df1_sum);
#res=lm(data=df1_sum, Parallel~Size);

#res;
#summary(res);

#ggplot(data=head(df,n=50), aes(x=x, y=y))+geom_point(alpha=.2)+geom_smooth(method="lm")
```

les étoiles définisssent le niveau de confiance (si la variable a un impact ou pas) r-SQUARED: un nombre entre 0 et 1 qui définit combien de bruit il reste, plus on s'approche de 1 moins on a de bruit).