Future

Garbuszus

2 Januar 2017

1 Future

Future ist ein R-Paket, um in R parallelisiert zu arbeiten. Warum ist das interessant?

Was meint parallelisiert und welche Alternativen gibt es?

1 Future

Warum ist R langsam?

- Schleifen sind in R langsam.
- R erzeugt keinen kompilierten Code.
- R läuft nur in einem Thread.

2 Single Threading

In computer programming, single threading is the processing of one command at a time. The opposite of single threading is multithreading. While it has been suggested that the term single threading is misleading, the term has been widely accepted within the functional programming community. (Wikipedia)

2 Single Threading

Computer arbeiten in Kernen und Programme in Threads auf den Kernen.

Grundsätzlich läuft ein Thread pro Kern, mit Hyperthreading, laufen auch zwei Threads pro Kern.

So kommt eine vier Kern CPU mit Hyperthreading auf acht Threads.

2 Single Threading

R ist ein eines Single Thread Programm. Alles in R läuft in einem Thread auf nur einem Kern. Das war lange Zeit ausreichend. Ein Thread war immerhin alles, was zur Verfügung stand.

Computer haben aber seit geraumer Zeit zwei und mehr Kerne. R ist das natürlich egal, es läuft nur in einem Thread.

Dadurch werden viele Ressourcen nicht genutzt und Auswertungen laufen entsprechend lange.

2 Single Threading

Wie kann man R beschleunigen?

- Schleifen vektorisieren (apply und lapply)
- Byte Code erzeugen (compile, Rcpp)
- R parallelisieren.

3 Parallelisierung

Neben Auswertungen auf einer CPU bzw. einem Single Threaded Programm, gibt es Multi Threaded Programme.

Programme, die nicht nur auf einer sondern auf zwei, drei oder allen CPUs/Threads laufen.

R vollständig in ein multithreaded Programm zu wandeln ist schwierig bis unmöglich, einzelne Auswertungsschritte zu entkoppeln ist jedoch möglich.

3 Parallelisierung

Möglichkeiten dazu sind vorhanden in den Bibliotheken

- parallel
- snow/snowfall
- foreach
- doParallel
- future

3 Parallelisierung

Was kann überhaupt parallelisiert werden?

Grundsätzlich jedes Problem, welches sich in Einzelprobleme zerlegen lässt.

Bspw. kann die Summe aus einem Vektor in Einzeloperationen zerlegt werden, die Stück für Stück bearbeitet werden und anschließend zusammengefasst werden.

4 Parallelisieren

Ein Beispiel für eine einzelne Funktion, die an einer einzelnen Aufgabe länger arbeitet ist `median()`

```
# Dichten ermitteln
x_median <- median(as.numeric(1:100000000))
```

Der Aufruf dauert auf meinem Rechner 1.82 Sekunden

```
x_median <- sum(as.numeric(1:100000000))
```

Die Bestimmung der Summe dauert dagegen mit 0.41 Sekunden nicht so lange.

4 Parallelisieren

`median()` berechnet nun den Median. Die Funktion braucht einen Vektor bestimmter Länge und arbeitet diesen ab.

Ein Großer Vektor führt zu langer Bearbeitungszeit.

`median()` ist schon byte-code. Die Funktion durch effizientere Programmierung zu beschleunigen ist nicht einfach.

Wir wollen uns daher etwas anders ansehen.

4 Parallelisieren

Warum future? Schauen wir uns die Syntax für den Median an. Gefolgt auf den Median, wird die Summe des Vektors berechnet und dann beides aufaddiert.

```
x_median <- median(as.numeric(1:100000000))
y_sum    <- sum(as.numeric(1:100000000))
z <- x_median + y_sum
```

4 Parallelisieren

```
z <- x_median + y_sum
```

In R warten wir nun die “relativ” langsame Berechnung von `x_median` ab, bevor die Summe `y_sum` gebildet wird und dann `z` bestimmt wird. Bis z bestimmt ist dauert es 2.25 Sekunden.

R wartet, weil es keinen weiteren Thread zur Verfügung hat, in dem bspw. die Summenbildung laufen könnte.

5 Future

Genau hier setzt `future` an. Future gibt R eine “Zukunft”.

```
# install.packages("future")
library(future)
plan(multiprocess)
```

5 Future

`future` bringt seine eigene Zuweisungspfeile mit.

Statt `y <- x` wird in `future` `y %<-% x` verwendet. Eine Zuweisung in `future` führt dazu, dass diese zunächst gesondert von den anderen Ausführungen durchgeführt wird.

5 Future

Ein Beispiel. Ohne Zukunft

```
a <- langsamerCode(obj) # lange warten
b <- langsamerCode(obj) # lange warten
c <- a + b #
1+1 # wird erst nach langem warten ausgefuehrt
```

5 Future

Gleiches Beispiel. Mit Zukunft

```
a %<-% langsamerCode(obj) # abschicken
b %<-% langsamerCode(obj) # kein warten, gleich weiter
c %<-% a + b # kein warten, abschicken
1+1 # wird sofort ausgefuehrt
```

5 Future

Was passiert?

Durch die Zuweisung in `future` führt R die erste Evaluation in einem eigenen Thread aus und blockt nicht die gesamte Auswertung.

Weil die zweite Auswertung ebenfalls in `future` steht, erfolgt auch diese Auswertung getrennt.

Beide Single Thread Funktionen, werden parallel abgearbeitet.

5 Future

Was passiert mit Objekten aus der Zukunft?

Sobald ein Objekt aus der Zukunft aufgerufen wird, wartet R mit der Evaluation, bis die Zukunft zur Gegenwart wird.

Sollen also die beiden Objekte weiter verarbeitet werden, muss zunächst die Evaluation abgeschlossen werden.

5 Future

Im Beispiel

```
# Dichten ermitteln
x_median %<-% median(as.numeric(1:100000000))
# Summe bilden
y_sum    %<-% sum(as.numeric(1:100000000))

# warten bis beides da ist, dann z ermitteln
z <- x_median + y_sum
```

Also alle <- durch %<-% ersetzen oder bringt das im Beispiel viel?

Hier dauert es 2.21 Sekunden.

Nein, im Beispiel bringt es nichts. Es dauert länger in R die einzelnen `future` Prozesse zu erzeugen als es braucht, eine optimierte Single Thread Funktion, wie `sum()` oder `median()` zu evaluieren.

5 Future

Wann bringt `future` also etwas? Wir nehmen ein anders Beispiel.

Dazu nehmen wir `npreg()` aus dem np-Paket (Hayfield and Racine (2008)). Und bestimmen eine nichtparametrische Regression für eine Reihe von Zufallsdaten.

```
# np laden
library(np)
```

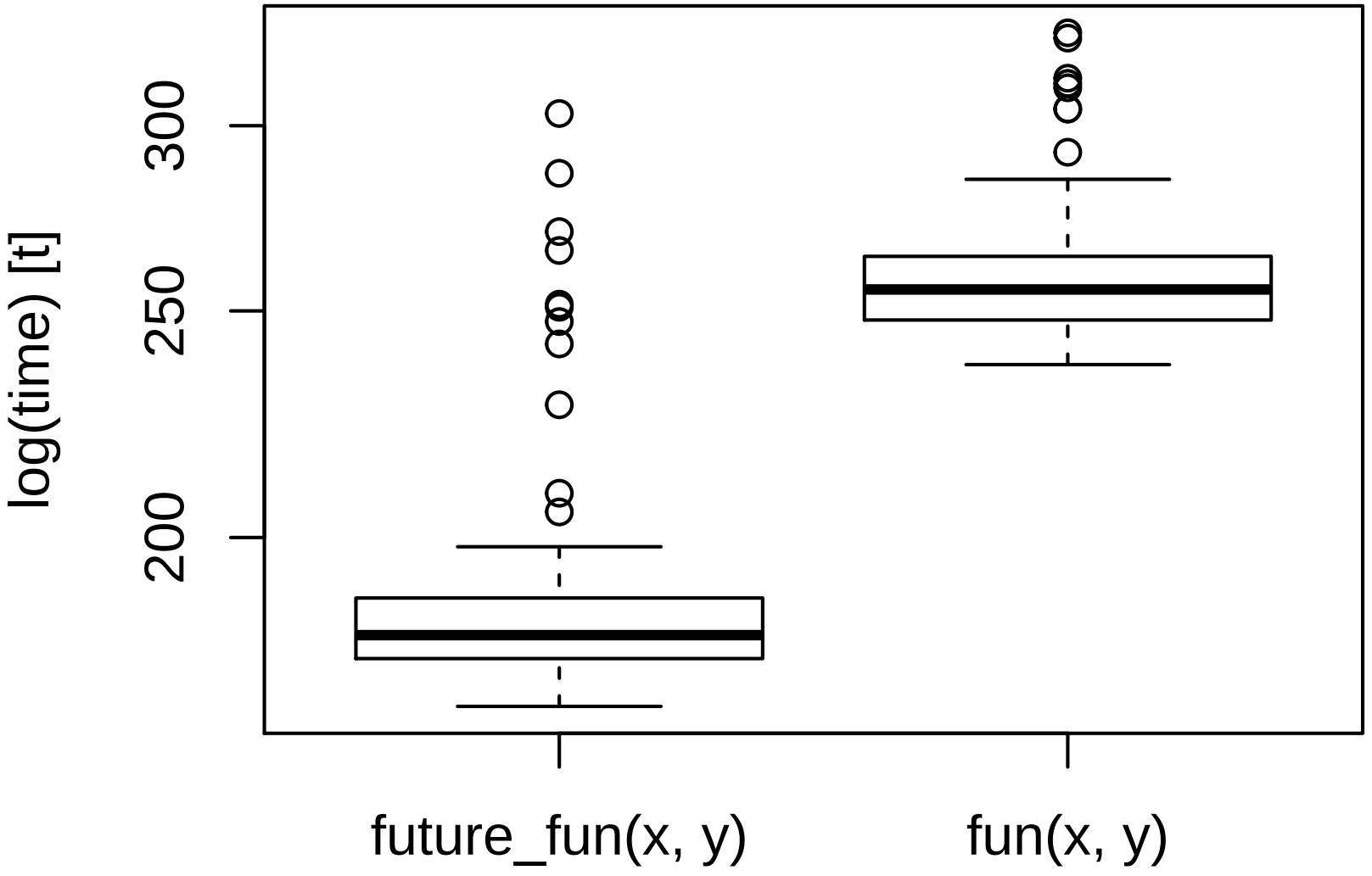
5 Future

```
future_fun <- function(x,y) {
  bw1 %<-% npreg(x~y)
  bw2 %<-% npreg(x~y)
  all.equal(bw1$bw, bw2$bw)
}

fun <- function(x,y) {
  bw1 <- npreg(x~y)
  bw2 <- npreg(x~y)
  all.equal(bw1$bw, bw2$bw)
}
```

5 Future

```
set.seed(12345)
x <- rnorm(100)
y <- abs(rnorm(100))

library(microbenchmark)
erg <- microbenchmark (
  future_fun(x,y), fun(x,y),
  times = 100
); erg
```

5 Future



5 Future

`future` kennt verschiedene Wege

Plan	führt zu
eager	sequenzieller Ausführung
lazy	Prüfung ob Objekt benötigt wird (sequentiell)
transparent	Ausgabe mit debug Informationen
mulitprocess	Paralleler Ausführung
multisession	Ausführung in einer weiteren R Sitzung (Windows)
multicore	Fork der aktuellen R Sitzung (Linux/Mac)
cluster	Ausführung auf mehreren Computern
remote	Steuerung per Fernzugriff

5 Future

`lazy` ermöglicht dabei, bestimmte Funktionen nur zu evaluieren, wenn diese tatsächlich benötigt werden. Im folgenden wird zB `b` gar nicht berechnet.

```
plan(lazy)

a %<-% sum(1:2); b %<-% sum(3:4)

for (i in 1:10) {
  if (i <= 10) { print(a) }     # nur a wird evaluiert
  else { print(b) }             # b wird evaluiert
}
```

5 Future

```
# kombinierte Plaene und nesting von Funktionen
plan(list(cluster, multiprocess))
a %<-% {
  c %<-% slow_sum(x[1:25])
  d %<-% slow_sum(x[26:50])
  c + d
}
b %<-% {
  c %<-% slow_sum(x[51:75])
  d %<-% slow_sum(x[76:100])
  c + d
}
```

5 Future

```
y <- a + b
```

5 Future

`future` ermöglicht also zunächst einmal den Umgang mit einem zukünftig verfügbaren Objekt bzw. ein Objekt zukünftig verfügbar zu machen.

Eine Zukunft muss dafür zunächst evaluiert werden. Bis dahin ist der Zustand des Objekts unerledigt kann aber auf erledigt wechseln und wird, bis dieser Zustand erreicht ist, weitere Aufrufe blocken.

5 Future

Ein entscheidender Vorteil von `future` ist, die für den Nutzer relativ einfache Implementierung. Natürlich kann noch immer irgendwas schief gehen, aber zumindest grundsätzlich kommen sich die einzelnen Prozesse nicht in den Weg.

Zu beachten dabei ist die Speicherauslastung, die kann im Einzelfall noch immer recht hoch sein.

6 foreach und apply

```
library(foreach)
library(parallel)

foreach(i=1:3) %do% sqrt(i)
foreach(i=1:3) %dopar% sqrt(i)

apply(as.array(1:3), 1, sqrt)
sapply(1:3, sqrt)
lapply(1:3, FUN = sqrt)
mclapply(1:3, FUN = sqrt) # nicht auf windows
```

7 Future!

Im Gegensatz zu `foreach` und `apply` muss keine neue Syntax gelernt werden bzw. groß umgeschrieben werden. Bekannte R Syntax ist zu 95% ausreichend.

Im Gegensatz zu `apply` müssen keine Listen-/Matrixobjekte vorliegen.

Im Gegensatz zu `foreach` ist das Ergebnis keine Liste.

7 Future!

```
# base R
for (i in 1:3) {
  for (j in 1:3) {
    ii <- i
    jj <- j
    print(ii, jj)
  }
}
```

7 Future!

```
# foreach
foreach(i = 1:3) %dopar% {
  foreach(j = 1:3) %dopar% {
    ii <- i
    jj <- j
    print(ii, jj)
  }
}
```

7 Future!

```
# future
for (i in 1:3) {
  for (j in 1:3) {
    ii %<-% i
    jj %<-% j
    print(ii, jj)
  }
}
```

8 Literatur

Hayfield, Tristen, and Jeffrey S. Racine. 2008. “Nonparametric Econometrics: The np Package.” *Journal of Statistical Software* 27 (5). http://www.jstatsoft.org/v27/i05/.