Анализ данных с R (III).

© С. В. Петров*, Е. М. Балдин†

*p2004r@gmail.com
†E.M.Baldin@inp.nsk.su

Эмблема **R** взята с официального сайта проекта `http://developer.r-project.org/Logo/`

Оглавление

Глава 10

Высокопроизводительные вычисления

Компьютеры становятся быстрее, места на диске всё больше, но темп с которым растёт объём данных и постоянно повышающаяся сложность их обработки всё равно *заставляет* думать как ускорить вычисления.

Традиционно для сложных вычислений используются кластеры. Многие слышали про список TOP500, но сейчас пора осознать что кластерные технологии потихоньку проникают и на обычные, можно сказать домашние, компьютеры. Почти у каждого процессора сейчас по несколько ядер. Так почему бы этим не воспользоваться? Но для начала постараемся понять как оценить выигрыш от подобных приёмов.

10.1. Анализ эффективности программы

Измерить скорость вычислений в **R** и, соответственно, оценить эффективность написанного кода можно несколькими способами:

- Использовать system.time() для простых измерений;
- Использовать Rprof() для профилирования написанного кода;
- Использовать Rprofmem() для профилирования использования памяти.

Для визуализации накопленных с помощью Rprof() данных можно использовать пакеты **profr** и **proftools**.

Применим средства профилирования к простой задаче. Пусть в ходе моделирования нам многократно требуется находить параметры линейной регрессии,

для чего воспользуемся функцией lm(). Единственный её недостаток заключается в том, что она делает много лишнего. Она просто слишком универсальна. Для простой линейной регрессии вполне достаточно воспользоваться уточнённым вызовом lm.fit().

Насколько же эффективней окажется прямой вызов? Попробуем с помощью обоих функций обработать набор макроэкономических показателей `longley`. Будем строить зависимость между числом работающих и остальными показателями. Для оценки проделав по 1000 вычислений за раз.

```
# Загружаем данные
> data(longley)
# Записываем профиль в файл lm.out
> Rprof("lm.out")
# Выполняем lm() 1000 раз
> invisible(replicate(1000,lm(Employed ~ .-1, data=longley)))
# Отключаем профилирование
> Rprof(NULL)
# Готовим данные для lm.fit()
> longleydm <- data.matrix(data.frame(longley))
# Записываем профиль в файл lm.fit.out
> Rprof("lm.fit.out")
# Выполняем lm.fit() 1000 раз
> invisible(replicate(1000,lm.fit(longleydm[,-7],longleydm[,7])))
# Отключаем профилирование
> Rprof(NULL)
```

Записанные в файлах данные профилирования можно проанализировать с помощью встроенной команды summaryRprof(). Например, для того чтобы выяснить время работы программы достаточно обратиться к переменной `sampling.time`:

```
> summaryRprof("lm.out")$sampling.time
[1] 6.42
> summaryRprof("lm.fit.out")$sampling.time
[1] 0.44
```

Данные профилирования можно отобразить и графически, с помощью пакета **profr**:

```
# Устанавливаем пакет (если нужно)
> install.packages("profr")}
> library("profr")
> plot(parse_rprof("lm.out"),main="Profile␣of␣lm()")
> plot(parse_rprof("lm.fit.out"),main="Profile␣of␣lm.fit()")
```

Из рис. 10.1 и 10.2 видно сколько времени проводит программа в каждой из функций. Неудивительно что функция lm.fit() работает в 14 раз быстрее.

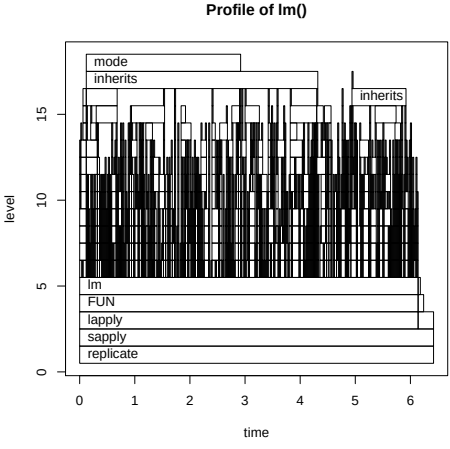

Рис. 10.1. Профилирование lm().

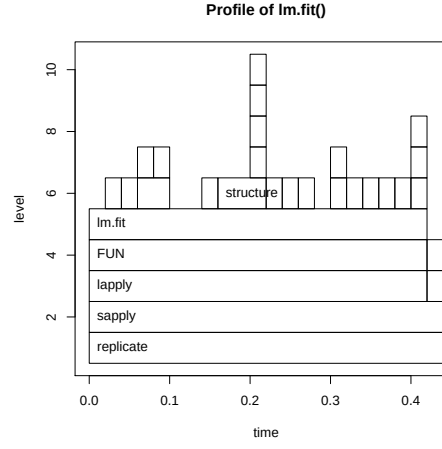

Рис. 10.2. Профилирование lm.fit().

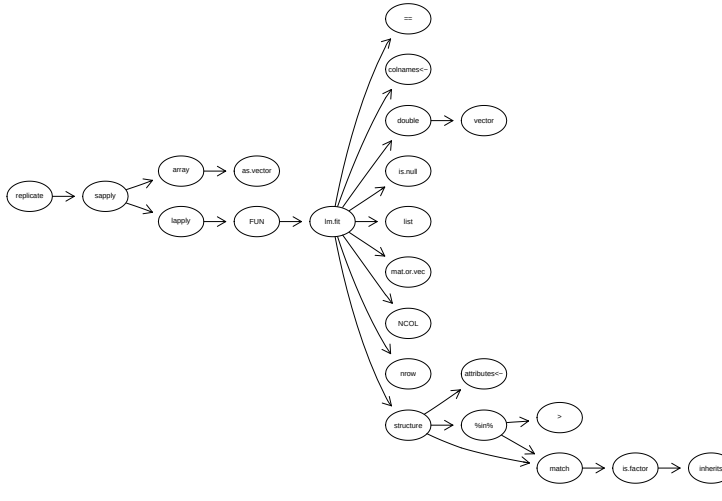

Рис. 10.3. Граф вызовов, полученный с помощью пакета **proftools**.

Для представления зависимостей вызовов в виде графа можно воспользоваться пакетом **proftools**. Для этого необходимо установить системный пакет **graphviz-dev**

```
=> aptitude install graphviz-dev
```

пакет **Rgraphviz** и сам **proftools**

```
> source("http://bioconductor.org/biocLite.R")
> biocLite("Rgraphviz")
> install.packages("proftools")}
```

Для отображения графа (рис. 10.3) необходимо выполнить следующие команды:

```
library("Rgraphviz")
library("proftools")
lmfitprod <- readProfileData("lm.fit.out")
plotProfileCallGraph(lmfitprod)
```

Для отладки потребления памяти нужно использовать специально модифицированную версию **R**, которая собрана с опцией `enable-memory-profiling`. При этом для анализа используется команда Rprofmem аналогично использованию Rprof.

Для контроля за избыточным потреблением памяти можно воспользоваться функцией tracemem. Она срабатывает каждый раз когда происходит копирование какого либо объекта.

10.2. Встроенные функции — ключ к ускорению

Достижение максимальной скорости во многом вопрос знания встроенных функций **R**. Их появление, как правило, обусловлено тем, что выполняемые ими функции встречаются в обработке данных часто. Подобный подход достаточно часто существенно сокращает усилия по вводу кода программы, а также повышает скорость выполнения кода, за счёт используемых во встроенных функциях оптимальных алгоритмов.

Очень часто встроенные функции реализованы на более низкоуровневом языке, который может предоставлять более широкий доступ к в возможностям оборудования. Поясним это небольшим примером. Для этого напишем программу складывающую числа от 1 до указанного с помощью обычного цикла, и сравним это с вариантом программы использующей встроенные функции **R**.

```
> mysum <- function(N) { a <- 0;
+    for (i in 1:N) a <- a + i;
+    return(a) }
> system.time(mysum(10000000))
```

```
   user  system elapsed
  7.456   0.024   7.482

> system.time(sum(as.numeric(seq(1,10000000))))
   user  system elapsed
  0.052   0.060   0.112
```

Разница очевидна: второй вариант подсчёта суммы работает более чем в 70 раз быстрее! То что сейчас было проделано носит название «векторизация». На такой способ расчётов трудно переключиться после императивного стиля программирования, но именно в векторизации заключается необычайно высокая продуктивность работы в **R**. Уменьшилось не только время выполнения программы, но и уменьшился её размер!

Возьмем простую тестовую задачу: «Найти распределение детерминанта матрицы 2x2 в которую занесены независимо и случайно изменяющиеся значения. Допустимые значения 0, 1, . . . , 9.» Она эквивалентна задаче найти все сочетания $ab - cd$, где a, b, c, d — это цифры.

Наивное императивное решение «с циклами» выглядит привычно и даже благодаря синтаксису **R** довольно «компактно»:

```
> dd.for.0 <- function()
+  {
+    val <- NULL
+    for (a in 0:9) for (b in 0:9) for (d in 0:9) for (e in 0:9)
+      val <- c(val, a*b - d*e)
+    table(val)
+  }
> system.time(dd.for.0())
   user  system elapsed
  0.196   0.000   0.195
```

Время от запуска к запуску слегка меняется и так как в подобных случаях распределение ненормальное, то лучше всего находить медиану нескольких попыток:

```
median(replicate(20, system.time(dd.for.0())["elapsed"]))
[1] 0.177
```

Попробуем добится от этого наивного варианта большего, например выделим память под расчёты сразу в начале программы:

```
> dd.for.1 <- function()
+  {
+    val <- double(10000)    # преалоцируем val
+    nval <- 0
```

```
+    for (a in 0:9) for (b in 0:9) for (d in 0:9) for (e in 0:9)
+      val[nval <- nval + 1] <- a*b - d*e
+    table(val)
+  }

> median(replicate(20, system.time(dd.for.1())["elapsed"]))
[1] 0.059
```

Улучшение очевидно: код ускорился более чем в три раза. Поскольку наши данные целые числа, то посмотрим что даст использование встроенной функции tabulate():

```
> dd.for.3 <- function()
+  {
+    val <- double(10000)
+    nval <- 0
+    for (a in 0:9) for (b in 0:9) for (d in 0:9) for (e in 0:9)
+      val[nval <- nval + 1] <- a*b - d*e
+    tabulate(val)
+  }
> median(replicate(20, system.time(dd.for.3())["elapsed"]))
[1] 0.057
```

Чуть-чуть улучшилось, но это все полумеры, делаем решительный шаг и вспоминаем что прародителем **R** был и язык APL. Запишем решение задачи как операцию над массивами:

```
> dd.fast <- function()
+  {
+    val <- outer(0:9, 0:9, "*")
+    val <- outer(val, val, "-")
+    tabulate(val)
+  }

> median(replicate(20, system.time(dd.fast())["elapsed"]))
[1] 0.001
```

Лучшее решение использующее циклы обойдено более чем в 50 раз, а «традиционное» почти в 200!

Что же делать если от цикла нельзя избавится? Есть вариант использовать среду **R** в которую встроена возможность компиляции кода.

Вариант сборки среды **R** с возможностью jit (just-in-time compilation) позволяет рассчитывать на ускорение кода содержащего циклы примерно в полтора раза.

Операции с матрицами являются в **R** такими производительными, потому что они опираются на процедуры библиотеки blass («basic linear algebra subprogram»). **R** может быть скомпилирована с различными вариантами реализации Blas: это и свободная библиотека Atlas (пакет **atlas3-base**), и платная Goto, и библиотеки от двух основных производителей процессоров Intel и AMD. Библиотеки не только имеют более производительный код, но и автоматически задействуют в вычислениях все имеющиеся ядра процессора персонального компьютера. Дополнительный прирост производительности можно получить настроив Atlas под конкретно используемый в расчётах персональный компьютер.

Независимо от наличия библиотеки BLAS можно поэкспериментировать с экспериментальным пакетом **pnmath0** от Люка Тьерни (Luke Tierney). Пакет можно найти по адресу `http://www.stat.uiowa.edu/~luke/R/experimental/`. Пакет заменят реализацию встроенных векторных функций **R** на параллельные варианты, используя Pthreads. Пока эта возможность не встроена в **R** и её придётся установить самостоятельно. Следует отметить, что параллельные вычисления будут активироваться только при достаточно длине векторов аргументов.

Если на компьютере установлена видеокарта которая поддерживает вычисления на своем GPU (Пока только CUDA и CUBLAS), то при установке пакета **gputools**, появляется возможность выполнять с очень высокой скоростью иерархический кластерный анализ, классификацию с обучением (По алгоритму SVM) и расчёт коэффициентов корреляции.

Несмотря на использование высокопроизводительных векторных операций и компиляции в режиме just-in-time, бывают моменты когда на счету каждый такт процессора. В этом случае есть два механизма оперативно встроить в расчет выполненный в среде **R** низкоуровневый код императивного языка программирования:

- Для простой вставки небольшого фрагмента кода **inline**;
- **Rcpp** — для облегчённого процесса интеграции сложного кода на C++.

Пакет **inline** предоставляет функцию cfunction(), умеющую автоматически встраивать код написанный на Fortran, C, C++. Для выполнения следующего простого примера на Fortran, естественно необходимо установить и загрузить сам **inline**:

```
# Не забудьте про отступы! Fortran такой Fortran.
> code <- "
+␣␣␣␣␣␣␣do␣i␣=␣1,␣n(1)
+␣␣␣␣␣␣␣␣x(i)␣=␣x(i)**3
+␣␣␣␣␣␣␣end␣do"
> cubefn <- cfunction(signature(n="integer", x="numeric"),
+                     code, convention=".Fortran")
> x <- as.numeric (1:10)
> n <- as.integer(10)
> cubefn(n,x)$x
```

```
[1]    1    8   27   64  125  216  343  512  729 1000
```

10.3. Параллелельные вычисления

Среда **R** работающая в 64х битном окружении практически не имеет ограничений по объёму обрабатываемых данных. Современным ответом в области вычисления с гигантскими объёмами данных является пакет **iterators** от REvolution Computing. В комплекте с возможностью поэлементно обработать структуру не помещающуюся в памяти крайне удачно сочетается второй пакет **foreach**. Данный пакет вводит возможность циклически обработать созданный итератор и вернуть суммарный результат. Отсутствие побочных эффектов позволяет выполнить оптимизируемую операцию параллельно.

Отсутствие побочных эффектов это именно то, что позволяет не заботится где выполняется та, или иная часть кода. Компьютеры не только стали мощнее и у них больше ядер. Компьютеров прежде всего стало *много* больше. Под рукой практически у каждого имеются 5–10 машин, многочисленные ЦПУ которых если посмотреть пристально никогда не загружены даже на 10% от своей возможности. Вся эта мощь доступна пользователю когда он использует **R**.

Кластерные вычисления настолько естественные для векторных операций **R**, что существует несколько способов их реализации в этой среде:

- **Rmpi** — это Message Passing Interface, является стандартом в области паралельных вычислений;
- **NWS** — это написанная на Python альтернативная реализация MPI;
- **snow** — высокоуровневая надстройка над MPI, PVM, NWS, sockets;
- **papply** — параллелизация функции apply через MPI;
- **multicore** — параллельные вычисления на многоядерных машинах.

Поскольку наша цель — это быстро и «глобально» задействовать мощь окружающих нас фактически простаивающих компьютеров для нашей же пользы, то сразу же воспользуемся высокоуровневым средством, а именно пакетом **snow**.

А что бы рассказ не был пустым теоретизированием попробуем решить реальную практическую задачу:

Задача Число телефонных пар, которые проходят рядом, и передают сигнал без помех ограничено. Безусловно влияет и длинна кабеля, и его ёмкость, и диаметр жил. Однако в номограммах оценки характеристик телефонной пары не предусмотрено свыше 25 ADSL пар в одном кабеле.

Известны нормативные документы российских провайдеров ADSL определяющме емкость одиночного кабеля в 18%. Скорее всего для кабелей небольшой

ёмкости всё же верен предел в 18 жил в одном кабеле занятых одновременно работающими ADSL модемами. А для кабелей более 100 пар ёмкости верно процентное ограничение.

В городе Нске например ёмкость ADSL сети Нсктелеком в этом году превысила 10 тыс. абонентов. Попробуем оценить какие проблемы встретит сеть при своем развитии.

Модель Город 300 тыс населения, в средней семье 3 человека определяет следующие исходные условия:

```
# кол-во абонентов услуги
> n.abonentov <- 10000
# ко-во домов
> n.domov <- 1000
# кол-во квартир в доме
> n.kvartir <- 100
# критическое число абонентов для ADSL в одном доме
> n.kritic <- 18
```

Получаем модель города в виде вектора, каждый элемент которого квартира помеченная номером дома

```
> gorod <- rep(1:n.domov, each = n.kvartir)
```

Поскольку вычисления ресурсоёмкие загрузим сразу библиотеку для паралельных вычислений основных функций пакета (предполагается, что эта экспериментальная библиотека уже установлена):

```
> library("pnmath0")
```

Делаем выборку случайных `n.abonentov` в векторе `gorod`.

```
> vyborka <- as.factor(gorod)[sample(1:length(gorod),
+             n.abonentov, replace= FALSE)]
```

Подсчитываем сколько в каждом доме попало абонентов, и строим гистограмму

```
> hist(as.numeric(tapply(rep(1,n.abonentov), vyborka, sum)))
```

Сколько домов испытывают трудности

```
> length(as.numeric(tapply(rep(1,n.abonentov),
+          vyborka, sum))[as.numeric(tapply(rep(1,n.abonentov),
+                           vyborka, sum))>n.kritic])
[1] 4
```

Сколько абонентов испытывают трудности

```
> sum(as.numeric(tapply(rep(1,n.abonentov),
+        vyborka, sum))[as.numeric(tapply(rep(1,n.abonentov),
+                    vyborka, sum))>n.kritic])
[1] 81
```

Это только одна реализация. Попробуем построить бутстреп процедуру и оценить какова доля домов в которых будет свыше `n.kritic` абонентов. Понадобится сделать не менее 10000 тысяч вычислительных экспериментов:

```
> nn <- 10000
> bstr.dom <- numeric(nn)
> bstr.abonent <- numeric(nn)
>  for (n in 1:nn) {
+    vyborka <- as.factor(gorod)[sample(1:length(gorod),
                 n.abonentov, replace= FALSE)]
+    rr <- as.numeric(tapply(rep(1,n.abonentov),vyborka , sum))
+    bstr.abonent[n] <- sum(rr[rr>n.kritic])
+    bstr.dom[n] <- length(rr[rr>n.kritic])
+  }
# Проблемные абоненты
> hist(bstr.abonent)
# Проблемные дома
> hist(bstr.dom)
```

Проведём эксперимент с подсчётом: сколько процентов абонентов и какое колво домов окажется с плохим качеством услуги при росте абонентской базы от 10000 до 20000 абонентов

Оформим функцию:

```
> my.boot.adsl <- function (n.abonentov) {
+  nn <- 10000
+  bstr.dom <- numeric(nn)
+  bstr.abonent <- numeric(nn)
+  for (n in 1:nn) {
+    vyborka <- as.factor(gorod)[sample(1:length(gorod),
+                      n.abonentov, replace= FALSE)]
+    rr <- as.numeric(tapply(rep(1,n.abonentov),vyborka , sum))
+    bstr.abonent[n] <- sum(rr[rr>n.kritic])
+    bstr.dom[n] <- length(rr[rr>n.kritic])
+  }
+  return(abonent=bstr.abonent, dom=bstr.dom)
+}
```

Рассчитаем оценку числа проблемных абонентов в диапазона от 10000 до 20000 размере абонентской базы:

```
> xx <- c(NA)
> for (n in 1:10) {
+  xx[n] <- my.boot.adsl(10000+(n*1000))
+}
```

Нормируем на размер абонентской базы

```
> xx.norm <- xx
> xx.norm[[1]] <- xx[[1]]/11000
> for (n in 2:10) {
>  xx.norm[[n]] <- xx[[n]]/(10000+(n*1000))
> }
```

Отобразим в виде боксплота

```
> boxplot(xx.norm,names=seq(11000,20000,by = 1000))
```

Данный модельный город предполагал наличие в городе только 100 квартирных домов. Повторим вычисления на данных о реальном количестве квартир в городе Нске.

Реальность Ввиду ресурсоёмкости вычислений напишем параллельную версию программы. Для этого загружаем библиотеки кластера

```
> library(snow)
```

Создаем кластер из двух узлов

```
 > cl <- makeCluster(c("localhost","localhost"), type = "SOCK")
```

Критическое число абонентов для ADSL в одном доме

```
> n.kritic <- 18
> clusterExport(cl, "n.kritic")
```

Получаем модель города в виде вектора, каждый элемент которого квартира помеченная номером дома. Обработаем реальные дома города Гродно. Загрузим список максимальных номеров квартир в доме

```
> Nsk <- na.omit(read.table("data_Nsk.txt"))
> Nsk.sort <- sort(t(Nsk))
```

Файл `data_Nsk.txt` — просто колонка числе, которую можно загрузить, например, тут: `http://www.inp.nsk.su/~baldin/data_Nsk.txt`.

Создадим модель города

```
> gorod <- unlist(mapply(rep,
+       1:length(Nsk.sort),Nsk.sort))
> clusterExport(cl, "gorod")
```

Отведём место под результаты оценки числа проблемных домов и числа проблемных абонентов. Для уменьшения накладных расходов при выделение памяти.

```
> nn <- 10000
> bstr.dom <- numeric(nn)
> clusterExport(cl,"bstr.dom")
> bstr.abonent <- numeric(nn)
> clusterExport(cl,"bstr.abonent")
```

Функция расчёта числа проблемных абонентов при случайном распределении

```
> my.boot.func <- function (n, n.abonentov) {
+  vyborka <- as.factor(gorod)[sample(1:length(gorod),
+                    n.abonentov, replace= FALSE)]
+  rr <- as.numeric(tapply(rep(1,n.abonentov),vyborka , sum))
+  bstr.abonent[n] <- sum(na.omit(rr[rr>n.kritic]))
+}
```

Рассчитаем оценку числа проблемных абонентов в диапазона от 3000 до 32000 размера абонентской базы

```
> xx <- cbind(sapply(1000+((1:10)*3000),
+                 function (n.abonentov) parSapply(cl,
+                      c(1:nn), my.boot.func, n.abonentov )))
```

Нормируем на размер абонентской базы

```
> xx.norm <- xx
> xx.norm[,1] <- xx[,1]/4000
> for (n in 2:10) {
+  xx.norm[,n] <- xx[,n]/(1000+(n*3000))
+ }
```

Отобразим в виде боксплота

```
> boxplot(xx.norm, names=seq(4000,31000,by = 3000))
```

Остановим кластер

```
> stopCluster(cl)
```

А в чём выигрыш? Приведённые выше вычисления тестировались на процессоре семейства Intel Core Duo модель T2050 с частотой 1.60 ГГц. При использовании распараллеливания время вычисления составляло 6470 секунд, а без оного 12754 секунд. Иными словами два ядра примерно в два раза лучше чем одно. **ЧТД**.

Настоящая реальность В соответствии с нормативными данными в номограммах для расчета помехозащищенности вообще не предусмотрен случай, когда число задействованных пар в одном кабеле превышает 25%. Это хорошо согласуется с тем, что работает даже в час пик только одна пятая часть всех абонентов. Иными словами полностью заполненный абонентами 100 парный кабель будет все таки работоспособен.

Однако есть два «но»:

1) реальный постсоветский кабель не держит более 18 нагруженных ADSL пар (в интернете есть нормативные документы российских провайдеров);

2) развитие IP – TV приводит абонентов с совсем другим подходом к использованию услуги. Ни о какой доле активных абонентов в одну пятую речи не идет. В утренние и вечерние часы телевизор смотрят 90%. Причем поток достигает максимальных скоростей и соответственно создаёт максимальные помехи для соседних в кабеле пар. И кол-во абонентов в самом лучшем случае не превысит 25%, даже при переходе к ADSL-2 (оптика на группу домов).

Получается что на настоящий день достигнув по агентурным данным размера абонентской базы в 15 тыс квартир, реальная компания Белтелеком в городе Гродно имеет долю клиентов имеющих периодически принципиально не устранимые проблемы в 30-38%. Причём каждый подключенный абонент ip-tv «съедает» ресурс сети как 5 абонентов передачи данных.