Wdrażanie efektywnej prostej średniej kroczącej

W ciągu ostatnich kilku dziesięcioleci zapotrzebowanie na moc obliczeniową stale rosło, wraz ze wzrostem ilości danych i bardziej złożonymi modelami. Oczywiste jest, że zminimalizowanie czasu potrzebnego na te obliczenia stało się ważnym zadaniem i że istnieją oczywiste problemy z wydajnością, które należy rozwiązać. Te problemy z wydajnością wynikają z niedopasowania ilości danych do istniejących metod analitycznych. Ostatecznie konieczna będzie fundamentalna zmiana w technikach analizy danych, ale na razie musimy zadowolić się poprawą wydajności naszych wdrożeń. R został zaprojektowany jako język interpretowany o wysokim poziomie ekspresji i to jest jeden z powodów, dla których brakuje mu dużej części precyzyjnej kontroli i podstawowych konstrukcji do obsługi kodu o wysokiej wydajności. Jak Arora wbija to w książkę, zredagowała Conquering Big Data with High Performance Computing, Springer, 2016: „Chociaż R jest wyraźnie językiem o wysokiej produktywności, niekoniecznie był to język o wysokiej wydajności”. Nierzadko zdarza się, że czas wykonania programu R jest mierzony w godzinach lub nawet w dniach. Wraz ze wzrostem ilości analizowanych danych czas wykonywania może stać się zbyt długi i często zdarza się, że naukowcy zajmujący się danymi i statystycy utknęli w tych wąskich gardłach. Kiedy tak się stanie i jeśli nie wiedzą zbyt wiele na temat optymalizacji wydajności, prawdopodobnie po prostu zadowolą się zmniejszoną ilością danych, co może utrudnić ich analizę. Jednak nie bój się; Programy w języku R mogą być powolne, ale dobrze napisane programy w języku R są zwykle wystarczająco szybkie, dlatego przyjrzymy się różnym technikom, których możesz użyć do zwiększenia wydajności kodu R. Ten rozdział nie ma na celu uczynienia z Ciebie eksperta w zakresie optymalizacji wydajności, ale raczej zawierają omówienie, które wprowadza Cię w ogromną liczbę technik, których można użyć podczas próby zwiększenia wydajności kodu. Przyjrzymy się wielu różnym technikom, z których każda może mieć rozdziały, a nawet książki poświęcone im, więc będziemy musieli spojrzeć na nie z bardzo wysokiego poziomu, ale jeśli będziesz ciągle ograniczany przez zasoby obliczeniowe, są one czymś będziesz chciał zajrzeć dalej. Niektóre z ważnych tematów omawianych w tym rozdziale są następujące:

\* Decydowanie, jak szybka musi być implementacja

\* Znaczenie korzystania z dobrych algorytmów

\* Powody, dla których R może czasami działać wolno lub nieefektywnie

\* Duży wpływ na wydajność, jakie mogą mieć małe zmiany

\* Pomiar wydajności kodu w celu znalezienia wąskich gardeł

\* Porównanie różnych implementacji między sobą

\* Maksymalne wykorzystanie możliwości komputera dzięki pracy równoległej

\* Poprawa wydajności dzięki współpracy z innymi językami

Wymagane pakiety

Pracowaliśmy już z niektórymi pakietami wymaganymi w tym rozdziale, takimi jak ggplot2 i lubridate. Pozostałe trzy pakiety zostały wprowadzone do testów porównawczych funkcji i porównania ich wydajności między sobą, a także do zaawansowanych technik optymalizacji, takich jak delegowanie i zrównoleglanie, które zostaną wyjaśnione w odpowiednich sekcjach. Aby móc odtworzyć wszystkie przykłady w tym rozdziale, potrzebujesz również działających kompilatorów dla kodu Fortran i C++. Zobacz Wymagane pakiety, aby uzyskać instrukcje dotyczące ich instalacji w systemie operacyjnym. Spójrzmy na poniższą tabelę przedstawiającą zastosowania wymaganych pakietów:

Pakiety : Powód

ggplo2 : Wysokiej jakości wykresy

lubridate : Łatwo przenosi daty

microbenchmark : Wydajność funkcji wzorcowych

str. 3

Zaczynając od dobrych algorytmów

Aby móc jasno przekazać idee zawarte w tutaj, najpierw muszę podać kilka prostych definicji. Mówiąc o algorytmie, mam na myśli abstrakcyjną specyfikację procesu. Kiedy mówię o implementacji, mam na myśli sposób, w jaki algorytm jest faktycznie programowany. Wreszcie, kiedy mówię o programie lub aplikacji, mam na myśli zbiór takich implementacji algorytmów współpracujących ze sobą. To powiedziawszy, łatwo jest zobaczyć, jak algorytm można zaimplementować na wiele różnych sposobów (na przykład jedna implementacja może korzystać z listy, a inna może używać tablicy). Każda z tych implementacji będzie miała inną wydajność i są one powiązane, ale nie równoważne, ze złożonością czasową algorytmu. Dla osób niezaznajomionych z ostatnim terminem każdy algorytm ma następujące dwie podstawowe właściwości

\* Złożoność czasowa: Ta właściwość odnosi się do liczby obliczeń, które algorytm musi wykonać, w odniesieniu do wielkości otrzymywanych danych wejściowych. Istnieją różne narzędzia matematyczne do pomiaru tej złożoności, z których najpowszechniejszym jest notacja Big-O, która mierzy najgorszy scenariusz dla algorytmu.

\* Złożoność przestrzeni: ta właściwość odnosi się do ilości pamięci wymaganej do wykonania algorytmu, ponownie w odniesieniu do rozmiaru danych wejściowych, które otrzymuje, i można ją również zmierzyć za pomocą tych samych narzędzi matematycznych. Powszechnie wiadomo, że nieefektywny algorytm zaimplementowany bardzo wydajnie może być o rząd wielkości wolniejszy niż wydajny algorytm zaimplementowany nieefektywnie. Oznacza to, że w większości przypadków wybór algorytmu jest znacznie ważniejszy niż optymalizacja implementacji.

Podczas oceny algorytmu należy wziąć pod uwagę wiele innych kwestii niż wspomniane wcześniej złożoności, takie jak wykorzystanie zasobów wydajności (na przykład przepustowość Internetu), a także inne właściwości, takie jak bezpieczeństwo lub trudność implementacji. Nie będziemy zagłębiać się w te tematy w tej książce. Jeśli jednak chcesz, aby Twój kod działał dobrze, musisz formalnie przestudiować struktury danych i algorytmy

str.4

Jak duży wpływ może mieć wybór algorytmu?

Obliczanie liczb Fibonacciego jest tradycyjnym przykładem przy nauczaniu rekurencji. Tutaj użyjemy go do porównania wydajności dwóch algorytmów, jednego rekurencyjnego i jednego sekwencyjnego. Jeśli ich nie znasz, liczby Fibonacciego są definiowane rekurencyjnie w sekwencji, w której następna jest sumą dwóch poprzednich, a dwie pierwsze liczby to jedynki (nasze przypadki bazowe). Rzeczywista sekwencja to 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144 i tak dalej. Nazywa się to ciągiem Fibonacciego i wykazuje interesujące właściwości, takie jak związek ze złotym podziałem, na który zdecydowanie powinieneś sprawdzić, jeśli nie wiesz, co to jest. Nasza funkcja fivbonaci\_recursive() przyjmuje pozycję liczby Fibonacciego, którą chcemy obliczyć jako n, ograniczoną do liczb całkowitych większych lub równych jeden. Jeśli jest to przypadek podstawowy, to znaczy, jeśli jest poniżej 1, po prostu go zwrócimy (nie, że jeśli obliczamy liczbę Fibonacciego na drugiej pozycji, nasza operacja n-2 wyniesie zero, co nie jest prawidłową pozycją, dlatego musimy użyć <= zamiast ==). W przeciwnym razie zwrócimy sumę wywołań rekurencyjnych do dwóch poprzednich za pomocą fibonacci\_recursive(n-1) i fibonacci\_recursive(n-2), jak pokazano w poniższym fragmencie kodu:

fibonacci\_recursive <- function(n) {

if(n <= 1) {return(n) }

return(fibonacci\_recursive(n-1) + fibonacci\_recursive(n-2)

Jak widać na poniższym fragmencie kodu, nasza funkcja działa zgodnie z oczekiwaniami. Co się jednak dzieje, gdy chcemy pobrać 35 lub 40 liczbę Fibonacciego? Jak możesz doświadczyć podczas uruchamiania tego kodu, im dalej liczba Fibonacciego jest od przypadków podstawowych, tym więcej czasu zajmie, a gdzieś w okolicach 30. pozycji zaczyna być zauważalnie wolniejsza. Jeśli spróbujesz obliczyć setną liczbę Fibonacciego, będziesz czekać długo, zanim otrzymasz wynik:

fibonacci\_recursive(1)

#> [1] 1

fibonacci\_recursive(2)

#> [1] 1

fibonacci\_recursive(3)

#> [1] 2

fibonacci\_recursive(4)

#> [1] 3

fibonacci\_recursive(5)

#> [1] 5

fibonacci\_recursive(35)

#> [1] 9227465

Dlaczego to się dzieje? Odpowiedź brzmi, że ten algorytm wykonuje dużo niepotrzebnej pracy, co czyni go złym algorytmem. Aby zrozumieć, dlaczego, przejdźmy mentalnie przez wykonanie algorytmu dla trzeciej i czwartej liczby Fibonacciego i utwórzmy odpowiednie drzewa wykonania, jak pokazano na poniższym diagramie:



Na poprzednim diagramie f(n) jest skrótem od fibonacci\_recursive(n) , dzięki czemu możemy zmieścić wszystkie obiekty w nim, a kolory są używane do pokazania, które wywołania funkcji są powtarzane. Jak widać, podczas wykonywania fibonacci\_recursive(3) , fibonacci\_recursive(1) wywołanie jest wykonywane dwukrotnie. Podczas wykonywania fibonacci\_recursive(4) to samo wywołanie jest wykonywane trzy razy. Ile razy będzie to wykonywane za fibonacci\_recursive(5) i fibonacci\_recursive(6)? To ćwiczenie dla ciebie, czytelniku, a jak się przekonasz, liczba ta rośnie wykładniczo. Aby być technicznie precyzyjnym, złożoność czasowa algorytmu wynosi O(2n), co jest tak złe, jak tylko można. Ponadto większość obliczeń jest całkowicie niepotrzebna, ponieważ są one powtarzane. Algorytm jest poprawny, ale jego wydajność jest jedną z gorszych. Jak wspomnieliśmy wcześniej, nawet jeśli zapewnisz najbardziej wydajną implementację tego algorytmu, będzie to znacznie wolniejsze niż bardzo nieefektywna implementacja bardziej wydajnego. Jeśli zaprojektujemy poprawny algorytm, który pozwoli uniknąć wykonywania wszystkich niepotrzebnych obliczeń, możemy to zrobić mają znacznie szybszy program i właśnie to robi następujący algorytm. Zamiast tworzyć drzewo rekurencyjnych wywołań, po prostu obliczymy liczby Fibonacciego w kolejności aż do tej, o którą jesteśmy proszeni. Po prostu dodamy poprzednie dwie liczby i zapiszemy wynik w tablicy, która będzie zawierała liczby całkowite. Określamy dwa przypadki podstawowe i kontynuujemy obliczenia, jak pokazano w poniższym fragmencie kodu:

fibanacci\_sequential <- funtion(n) {

if (n <= 2) { return(1)

f <- integer(n)

f[1] <- 1

f[2] <- 1

for (i in 3:n) {

f[i] <- f[i-2] + f[i-1]

}

return(f[n])

 Jak widać, każda liczba jest obliczana tylko raz, co jest najbardziej wydajnym algorytmem, jaki możemy zaprojektować dla tego problemu. Pozwala to uniknąć całego narzutu związanego z algorytmem rekurencyjnym i pozostawia nam liniową złożoność czasową O (n). Nawet jeśli kodujemy ten algorytm bez dbania o optymalizację wydajności, jego czas wykonania będzie szybszy o wiele rzędów wielkości. Za pomocą tego algorytmu możemy w rzeczywistości obliczyć 1476 liczbę Fibonacciego, która jest największą, na jaką pozwala architektura wewnętrzna R. Jeśli spróbujemy obliczyć 1477. liczbę Fibonacciego, otrzymamy nieskończoność (Inf) jako odpowiedź ze względu na mechanizmy używane przez R do przechowywania liczb całkowitych, co jest tematem, którym nie będziemy się zajmować. Co więcej, obliczenia dla 1476 liczby Fibonacciego są prawie natychmiastowe, co pokazuje, jak ważne jest wybranie dobrego algorytmu, zanim zacznie się go optymalizować:

fibonacci\_sequemtial(1476)

# [1] 1.306989e + 308

fibonacci\_sequemtial(1477)

#[1] Inf

Na koniec zauważ, że osiągnęliśmy wzrost szybkości kosztem wykorzystania większej ilości pamięci. Algorytm rekurencyjny odrzucał każdą liczbę Fibonacciego po jej obliczeniu, podczas gdy algorytm sekwencyjny przechowuje każdą w pamięci. W przypadku tego konkretnego problemu wydaje się to być dobrym kompromisem. Kompromis między czasem i przestrzenią jest powszechny w optymalizacji wydajności. Teraz, gdy zobaczyliśmy, jak ważny może być wybór algorytmu, w dalszej części rozdziału będziemy pracować z jednym algorytmem i skupimy się na optymalizacji jego implementacji. Pozostaje jednak kwestia, że ​​wybór wydajnego algorytmu jest ważniejszy niż efektywne jego wdrożenie.

str. 7

Jak szybko jest wystarczająco szybko?

Załóżmy, że wybrałeś dobry algorytm i zaimplementowałeś go bez zbytniego dbania o optymalizację, jak to zwykle bywa przy pierwszych próbach. Czy warto poświęcić czas na jego optymalizację? Optymalizacja wydajności może być bardzo kosztowną czynnością. Nie możesz próbować optymalizować swojego kodu, chyba że musisz. Twój czas jest cenny i prawdopodobnie lepiej spędzić go na robieniu czegoś innego. Powiedzmy, że z jakiegoś powodu musisz przyspieszyć implementację. Pierwszą rzeczą, o której musisz zdecydować, jest to, jak szybko jest wystarczająco szybko. Czy twój algorytm musi po prostu zakończyć się w ciągu kilku godzin zamiast kilku dni, czy też musisz zejść do poziomów mikrosekund? Czy jest to bezwzględny wymóg, czy po prostu powinieneś wykonać najlepszą pracę, jaką możesz w określonym czasie? Są to ważne pytania, które należy rozważyć przed optymalizacją kodu, a czasami rozwiązaniem nie jest nawet optymalizacja. Nierzadko zdarza się, że klienci wolą wydawać więcej pieniędzy na używanie pewnego rodzaju zasobów w chmurze w celu rozwiązania problemu z wydajnością, zamiast spędzać cenny czas na optymalizowaniu wydajności algorytmu, zwłaszcza jeśli mogą zapewnić większą wartość biznesową, robiąc coś innego. Oprócz wspomnianego wcześniej kompromisu między czasem maszynowym a czasem ludzkim, przy podejmowaniu decyzji o optymalizacji implementacji algorytmu należy wziąć pod uwagę inne kwestie. Chcesz, aby Twój kod był czytelny? Czy chcesz, aby Twój kod był dostępny? Często zdarza się, że bardziej wydajny kod jest również trudniejszy do zrozumienia. Co więcej, jeśli tworzysz kod, który jest wykonywany równolegle, nałoży to szereg ograniczeń na typ systemów, które mogą go wykonać, i musisz o tym pamiętać. Proponuję ograniczyć do minimum liczbę optymalizacji; dzięki temu osiągniesz swój cel dotyczący tego, jak szybko musi działać implementacja, i nie rób więcej. Proces będzie prosty: znajdź najważniejsze wąskie gardło, usuń je (lub przynajmniej zmniejsz jego wpływ), sprawdź, czy Twoja implementacja jest wystarczająco szybka, a jeśli nie, powtórz. W tym rozdziale przejdziemy przez ten cykl kilka razy i chociaż z perspektywy czasu wydaje się to łatwe, w rzeczywistości może być dość trudne, szczególnie w przypadku złożonych algorytmów.

str. 8

Obliczanie prostych średnich kroczących jest nieefektywne

Algorytm, z którym będziemy pracować w dalszej części rozdziału, nazywa się prostą średnią ruchomą (SMA). Jest to bardzo dobrze znane narzędzie do przeprowadzania analizy technicznej szeregów czasowych, szczególnie dla rynków finansowych i handlu. Ideą SMA jest to, że obliczysz średnią na każdy punkt w czasie, patrząc wstecz na określoną liczbę okresów. Na przykład, powiedzmy, że patrzysz na szeregi czasowe minuta po minucie i obliczysz SMA (30). Oznacza to, że przy każdej obserwacji w swoim szeregu czasowym weźmiesz obserwacje odpowiadające poprzednim 30 minutom od rozpoczęcia określonej obserwacji (30 obserwacji wstecz) i zapiszesz średnią z tych 30 obserwacji jako SMA (30 ) wartość dla tego punktu w czasie.

Na późniejszym diagramie możesz zwizualizować ideę SMA. Diagram przedstawia monotoniczne szeregi czasowe, które wzrastają o jedną jednostkę wartości dla każdej jednostki czasu, z których obie zaczynają się od jednej (to znaczy, że jej wartość wynosi 1 w czasie 1, 2 w czasie 2 itd.), Wzdłuż z pewnymi liczbami otaczającymi grupę obserwacji, zostaną podjęte obliczenia SMA. Jak widać, w przypadku SMA (3) ostatnie trzy elementy otrzymujemy w każdym punkcie szeregu czasowego; podobnie dla SMA (4) otrzymujemy cztery ostatnie elementy. Kiedy obliczasz średnią dla podzbioru elementów na rysunku, otrzymasz liczby w lewym górnym rogu, które odpowiadają określonym obliczonym szeregom czasowym SMA. W szczególności dla takich szeregów czasowych, dla przypadku SMA (3), wynikiem jest NA, NA, 2, 3, 4, 5, 6, 7 i 8, a dla przypadku SMA (4) wynik to NA, NA, NA, 2,5, 3,5, 4,5, 5,5, 6,5 i 7,5. Istnieje kilka następujących właściwości, na które powinniśmy zwrócić uwagę na temat SMA:

* Po pierwsze, należy zauważyć, że zarówno SMA(3), jak i SMA(4) są szeregami zawierającymi taką samą liczbę obserwacji jak oryginalne szeregi czasowe, w tym przypadku 9.
* Po drugie, zauważ, że oba zaczynają się liczbą NA równą liczbie parametru SMA minus jeden. Dzieje się tak, ponieważ w przypadku SMA (3) w czasie 2 nie mamy trzech obserwacji wstecz, mamy tylko dwie. Dlatego NA jest używany do wskazania, że ​​w tym momencie nie można było obliczyć SMA (3). To samo wyjaśnienie dotyczy wszystkich innych wartości NA.
* Po trzecie i wreszcie, zauważ, że za każdym razem, gdy przesuwamy się o jedną jednostkę czasu, dodajemy jedną obserwację i usuwamy kolejną obserwację (ogon) z bieżącego podzbioru.

str. 9

Symulacja szeregów czasowych

Oczywiście zbierasz dane z rynków kryptowalut, odkąd zaimplementowałeś własną wersję systemu obiektowego, który opracowaliśmy w poprzednim rozdziale, prawda? Tylko żartuję. Jeśli tak, to prawdopodobnie nie są wystarczające dane do tego, co zrobimy w tym rozdziale, więc oto mały fragment kodu, który będzie symulował dwie serie czasowe dla ceny Bitcoin i Litecoin w dolarach amerykańskich. Struktura danych jest podobna do tej zastosowanej w poprzedniej części, dzięki czemu kod, który tu opracowujemy, jest przydatny również w tym systemie. Nie będziemy zagłębiać się zbytnio w działanie tej funkcji, ponieważ w tym momencie powinno być dla Ciebie jasne, z wyjątkiem wskazania, że ​​używamy funkcji time\_to\_timesamp(. TomeStamp(), którą opracowaliśmy poprzednio , i że funkcja simulate\_prices()wykorzystuje model kwadratowy górę symulacji ARIMA. Spójrzmy na następujący kod:

source(„../chapter-08/cryptourrencies/utilities/time-stamp.R”\_

library(lubridate)

N <- 60 \* 24 \* 365

simulate\_market <- funtion(name, symbol. now, n , base, sd, x ) {

dates <- seq(now- minutes(n-1), nowm by = „min”)

ts <- unlist(lapply(lapply (

dates, times\_to\_timestamp.TimeStamp), unclass))

price\_usd <- simulate\_prices(n, base, sd, x)

data <- data.frame(timestamp – ts, prie\_usd = price\_usd)

data$name <- name

data$symbol <- symbol

return(data)

simulate\_prices <- funtion(n, base, sd, x) {

ts <- arima.sim(list(15, 15,15(, n = n , sd = sd)

quadratic\_model <- base + (x-1) \* base / (n^2) \* (1:n)^2

return(as.numeri(ts+quadrati\_model))

now <- Sys.time()

btc <- simulate\_market(„Bitcoin, „BTC”, now, N , 8000, 8 ,2)

bltc <- simulate\_market(„Litecoin, „LTC”, now, N , 80, 8 ,2)

data <- rbind(bt, ltc)

data <- data[order(data$timestamp) , ]

write.sv(data, „./data.csv”, row.names = FALSE)

Zauważ, że parametry użyte do wywołania funkcji simulate\_market() starają się przypominać co widać obecnie w cenach Bitcoin i Litecoin, ale pamiętaj, że jest to bardzo prosty model, więc nie oczekuj, że będzie on zachowywał się jak rzeczywiste szeregi czasowe cen tych aktywów. Na koniec symulujemy 525 600 obserwacji dla każdego zasobu, co w przybliżeniu odpowiada liczbie minut w roku (N <- 60 \* 248 365, która obejmuje sekundy na godzinę, godziny dziennie i dni w roku). Oznacza to, że symulujemy dane minuta po minucie. Aby zwizualizować ceny Bitcoinów, które symulowaliśmy, możesz użyć następującego kodu. Po prostu tworzy jeden wykres, który wykorzystuje próbkę 1000 elementów przez cały rok (więcej niż to jest niepotrzebne, ponieważ nie będziesz w stanie dostrzec więcej punktów, a to spowolni obliczenia); tworzony jest również inny wykres, który pokazuje efekt powiększenia danych do pierwszej godziny:

s <- sample(1:nrow(btc), 1000)

plot(btc[s[order(s)], „price\_usd”], xlab = „Minutes”, ylab=”Price” , xaxt = ‘n’)

title(main=”Bitcoin price simulation for 1 year”)

lines(btc[s[order(s)], „proce\_usd”)

plot(btc[1:60, „price\_usd”], xlab = „Minutes”, ylab=”Price” , xaxt = ‘n’)

title(main=” Bitcoin price simulation for 1 hour”)

lines(btc[1:60, „price\_usd”[)

Jak widać, obserwując całoroczną symulację, obserwuje się silny trend wzrostowy, ale jeśli powiększysz do mniejszego przedziału czasowego, zobaczysz sporą różnicę cen, która pozwala na użyteczne implementacje SMA

str 11

Nasza pierwsza (bardzo nieefektywna) próba SMA

Jak wspomniano wcześniej, przez pozostałą część rozdziału będziemy pracować z implementacjami SMA. Aby je rozróżnić, będziemy zmieniać nazwę funkcji dla każdej kolejnej implementacji, zaczynając od sma\_slow\_1(). Wszystkie implementacje SMA otrzymają następujące parametry:

**period** : Aby określić, ile obserwacji ma być używanych dla SMA.

**symbol**: Aby oznaczyć składnik aktywów, dla którego chcemy wykonać obliczenia. W tym przykładzie opcje będą for lub for. Jednak gdy samodzielnie uruchomisz system, będziesz mógł go rozszerzyć o dowolny symbol kryptowaluty, który sobie zażyczysz.

**data** : Rzeczywiste dane zawierające szeregi czasowe cen dla każdego zasobu.

Zrobimy dwa założenia, że ​​kolumna data - timestamp jest w porządku rosnącym i nie mamy luk w szeregach czasowych, co oznacza, że ​​mamy dane cenowe z każdej minuty. To pozwala nam pominąć wszelkie procedury zamawiania i sprawdzić, czy SMA powinno zawierać wewnętrznie NA, gdy nie są dostępne żadne dane. Zauważ, że oba te założenia są spełnione przez naszą symulację danych. Teraz wyjaśnimy, jak sma\_slow\_1() działa. Zauważ, że jest to bardzo nieefektywna implementacja i zdecydowanie powinieneś unikać programowania w ten sposób. Są to jednak typowe błędy popełniane przez ludzi i będziemy je usuwać jeden po drugim, mierząc ich wpływ na szybkość kodu. Zobaczmy, jak to się robi, wykonując następujące kroki:

1. Najpierw tworzymy pustą ramkę danych nazwaną result, który zawiera pojedynczą kolumnę o nazwie sma.

2. Następnie wykonujemy pętlę po wszystkich wierszach danych; oznacza koniec lub prawy koniec rozważanego interwału SMA.

3. Tworzymy liczbę całkowitą position, która jest taka sama jak end za każdym razem, gdy zaczynamy pętlę, a także obiekt sma, który będzie zawierał rzeczywiste obliczenia SMA dla pozycji końcowej, liczbę całkowitą n\_accumulated, która śledzi liczbę zgromadzonych obserwacji oraz ramkę danych period\_prices, która zawiera jedną kolumnę do przechowywania cen dla bieżącej kalkulacji SMA.

4. Następnie sprawdzamy, czy obserwacja przy bieżącym end odpowiada temu symbol, co nas interesuje. Jeśli tak nie jest, po prostu ignorujemy tę iterację, ale jeśli tak, będziemy kumulować perdiod\_prices zaczynając od pozycji end (pamiętaj, że position równa się end i temu punktowi) i cofając się do liczby skumulowanych cen jest równa period interesującej nas lub aktualnej position mniejszej niż 1 (co oznacza, że ​​jesteśmy na początku szeregu czasowego). Aby to zrobić, używamy pętli while, która sprawdza warunek wspomniany wcześniej, zwiększa n\_acumulated, gdy zostanie znaleziona obserwacja z tym samym symbol , a jej dane są dołączane do ramki danych period\_prices, i zwiększa position niezależnie od tego, czy obserwacja była przydatna, abyśmy nie utknęli.

5. Po zakończeniu pętli while wiemy, że albo zgromadziliśmy liczbę cen równą period, która nas interesuje, albo napotkaliśmy początek szeregu czasowego. W pierwszym przypadku obliczamy średnią takich cen poprzez iterację po ramce danych period\_prices i przypisujemy ją jako wartość sma dla aktualnej pozycji end. W drugim przypadku po prostu rejestrujemy wartość NA, ponieważ nie byliśmy w stanie obliczyć pełnego SMA. Spójrz na następujący fragment kodu:



Jeśli implementacja wydaje się skomplikowana, to dlatego, że tak jest. Gdy zaczniemy ulepszać nasz kod, zostanie on oczywiście uproszczony, co ułatwi jego zrozumienie.

6. Teraz chcemy faktycznie zobaczyć, że to działa. Aby to zrobić, wprowadzamy plik sma-slow.R do pamięci (która zawiera wszystkie powolne implementacje), a także dane, jak pokazano w poniższym fragmencie kodu:

source(„./sma-slow.R”)

data\_original <- read.csv(„.data.csv”)

Zwróć uwagę, że bierzemy tylko pierwsze 100 obserwacji, które odpowiadają 50 minutom akcji cenowej Bitcoin (pamiętaj, że te 100 obserwacji zawiera tylko 50 dla Bitcoin; pozostałe 50 dotyczą Litecoina). Widzimy, że SMA (5) dla Bitcoin ma sens, w tym pierwsze cztery NA (możesz sprawdzić liczby ręcznie, ale pamiętaj, aby wykorzystać dane i wyniki do własnej symulacji danych):

data <- data\_original[1:100, ]

symbol <- „BTC”

period <- 5

sma-1 <= sma\_slow\_1(period, symbol, data)

sma\_1

#> sma

#> 1 NA

#> 2 NA

#> 3 NA

#> 4 NA

#> 5 7999,639

#> 6 7997.138

#> 7 8000.098

#> 8 8001.677

#> 9 8000.633

#> 10 8000.182

(Obcięte wyjście)

Zanim zrozumiemy, jak naprawić ten kod, musimy zrozumieć, dlaczego R może być powolne, a także jak zmierzyć wpływ, jaki wywieramy, gdy go ulepszamy.

str. 14

Zrozumienie, dlaczego R może być powolny

Zrozumienie, dlaczego język programowania może być powolny, jest podstawową umiejętnością potrzebną do zwiększenia szybkości jego implementacji. Na każdą implementację w dowolnym języku programowania ma podobny wpływ czas algorytmu i złożoność pamięci, ponieważ są one algorytmami, a nie właściwościami implementacji. Jednak sposób, w jaki języki obsługują określone implementacje, może się znacznie różnić i na tym się teraz skupimy. W przypadku R ludzie często znajdują cztery główne wąskie gardła:

\* Niezmienność obiektu

\* Zinterpretowane dynamiczne typy

\* Procesy związane z pamięcią

\* Procesy jednowątkowe

W żadnym wypadku ta lista nie jest kompletna ani nie występuje w każdej implementacji. To tylko najczęstsze wąskie gardła, jakie widziałem, a które po naprawieniu spowodowały największą liczbę ulepszeń szybkości. Często są to dobre punkty wyjścia, ale każda implementacja jest inna, więc bardzo trudno jest zasugerować ogólne zasady optymalizacji wydajności i należy o tym pamiętać.

str. 15

Niezmienność obiektu

Poprawa szybkości implementacji języka R niekoniecznie wiąże się z zaawansowanymi technikami optymalizacji, takimi jak zrównoleglenie. Rzeczywiście, istnieje wiele prostych poprawek, które choć nie zawsze są oczywiste, mogą sprawić, że R będzie działał znacznie szybciej. Największym wąskim gardłem, jakie ludzie napotykają w przypadku R, jest brak zrozumienia właściwości niezmienności obiektu oraz kosztów ogólnych poniesionych podczas wykonywania kopii takich obiektów. Samo zajęcie się tym może przynieść radykalną poprawę wydajności i nie jest to zbyt trudne, gdy zrozumiesz, jak to zrobić. To dobry kandydat do rozpoczęcia poszukiwań optymalizacji. Jako przykład niektórych problemów, które mogą się pojawić, załóżmy, że masz nazwaną tablicę liczb. Teraz załóżmy, że chcesz zaktualizować pierwszy element programu to be, jak pokazano w poniższym fragmencie kodu:

a[1] <- 10

To zadanie jest o wiele bardziej złożone, niż się wydaje. W rzeczywistości jest realizowany za pośrednictwem funkcji zastępczej `”[<-„1` poprzez to wywołanie i przypisanie:

a <- `”[<-„` (a, 1, value = 10)

Na początku może się wydawać, że jest to bardzo dziwna składnia, ale pamiętaj, że jak widzieliśmy,że możemy mieć ciągi znaków, które reprezentują obiekty, w tym funkcje, tak jak w tym przypadku. Części linii `”[<-„` to w rzeczywistości nazwa funkcji wywoływanej z parametrami, a,1 i value = 10. Jeśli wykonasz poprzednie dwie linie, powinieneś otrzymać ten sam wynik; to jest pierwszy element w a bycia równym 10. To, co faktycznie się dzieje, to tworzenie wewnętrznej kopii; pierwszy element takiego obiektu zostanie zmieniony na 10 , a wynikowy obiekt zostanie ponownie przypisany do a. Mimo że po prostu zmieniamy tylko jeden element tablicy, w rzeczywistości cały wektor jest ponownie obliczany. Im większy wektor, tym gorszy problem, a to może znacznie spowolnić implementację. Jest jeszcze gorzej, gdy używasz ciężkich struktur danych, takich jak ramki danych. Języki, które pozwalają na mutabilność, takie jak Fortran lub C++, po prostu zmienią określoną wartość w tablicy zamiast tworzyć nową kopię pełnej tablicy. Dlatego często zdarza się, że kod, który byłby w porządku w innych językach, generuje bardzo duże i często niepotrzebne narzuty, gdy jest programowany w podobny sposób w R.

str. 16

Zinterpretowane typy dynamiczne

Drugim najważniejszym wąskim gardłem, które ludzie znajdują, jest natura języka R, który jest językiem interpretowanym i dynamicznie wpisywanym na maszynie. Oznacza to, że w dowolnym wierszu programu obiekt może być liczbą całkowitą, w następnym wierszu może to być ramka danych, następnie łańcuch znaków i może to być lista ramek danych dwa wiersze później. Jest to naturą braku ustalonych typów obiektów, a ponieważ interpreter nie może z góry wiedzieć, jak obsługiwać takie obiekty, ponieważ za każdym razem mogą one być zupełnie inne, musi sprawdzać typ obiektu za każdym razem, gdy chce zastosować jakieś rodzaj operacji na nim. Jest to trochę przesadzone, ale chodzi o to, że skoro istnieje możliwość zmiany typu obiektu, należy go stale sprawdzać. Zobaczymy, jak uniknąć niektórych z tych kontroli, aby zwiększyć wydajność, ale aby poradzić sobie z interpretowaną i dynamicznie wpisywaną naturą, będziemy musieli skorzystać z innych języków programowania, takich jak Fortran lub C++, co pokażemy w dalszej części rozdziału . Te języki naprawiają typ obiektu podczas jego tworzenia i jeśli spróbujesz go zmienić w pewnym momencie, program zgłosi błąd. Może to być postrzegane jako niepotrzebne ograniczenie, ale w rzeczywistości może być bardzo potężne, gdy komunikuje zamiar jakiegoś kodu, a także pozwala kompilatorom na zapewnienie potężnych optymalizacji obsługi takich obiektów.

str. 16

Procesy związane z pamięcią

Trzecim najważniejszym wąskim gardłem, które ludzie odkrywają, jest to, że R musi mieć wszystkie obiekty w pamięci. Oznacza to, że komputer używany do analizy musi mieć wystarczającą ilość pamięci RAM, aby pomieścić jednocześnie wszystkie dane, a także obiekty pośrednie i wynikowe, i należy pamiętać, że ta pamięć RAM jest współdzielona z wszystkimi innymi aplikacjami działającymi na komputerze. Jeśli R nie ma wystarczającej ilości pamięci RAM, aby pomieścić każdy obiekt w pamięci, system operacyjny wykona operację zamiany, która w R będzie wyglądać tak, jakbyś miał wszystkie dane w pamięci, ale dane zostaną zapisane i odczytane z dysku twardego w rzeczywistość. Czytanie i pisanie z dysków twardych jest o rząd wielkości wolniejsze niż wykonywanie równoważnych operacji w pamięci, a R nie poinformuje Cię, że tak się dzieje, ponieważ naprawdę nie może (robi to system operacyjny). Aby wykryć, że tak się dzieje, należy zwrócić uwagę na narzędzie dostarczane przez system operacyjny do monitorowania zasobów systemu. Mimo że jest to trzecie wąskie gardło na liście, kiedy się zdarza, jest zdecydowanie najbardziej szkodliwe, ponieważ mamy wąskie gardło wejścia / wyjścia dysku na szczycie wąskiego gardła pamięci. Kiedy napotkasz ten problem, będziesz w stanie stwierdzić, ponieważ R wydaje się zawieszać lub nie reagować. Jeśli to ci się przytrafia, zdecydowanie powinieneś poszukać sposobów, aby to wyeliminować. Jest to trzecie miejsce na liście, ponieważ nie występuje tak często jak poprzednie dwa, a nie dlatego, że ma mniejszy wpływ.

str. 17

Procesy jednowątkowe

Czwartym najważniejszym wąskim gardłem, z którym spotykają się ludzie, jest fakt, że język R nie ma wyraźnych konstrukcji dla paralelizmu. Natychmiastowa instalacja R nie może korzystać z wielu procesorów i nie ma znaczenia, czy zainstalujesz R na potężnym serwerze z 64 rdzeniami procesora, R użyje tylko jednego z nich. Sposobem na rozwiązanie tego problemu jest wprowadzenie równoległości w implementacjach. Jednak zrobienie tego wcale nie jest łatwym zadaniem. W rzeczywistości poważne wysiłki związane z równoległością wymagają głębokiej wiedzy o sprzęcie i oprogramowaniu i często zależą od konkretnego sprzętu używanego do wykonania implementacji. Chociaż jest to bardzo trudne, a może nawet z tego powodu, R ma wiele pakietów, których celem jest zapewnienie równoległych rozwiązań dla określonych funkcji języka R. Istnieje kilka ogólnych pakietów, których możesz użyć do tworzenia własnych równoległych implementacji, co zobaczymy w dalszej części tego rozdziału, ale zdecydowanie nie jest to pierwsze miejsce, w którym należy szukać ulepszeń wydajności. Teraz, gdy rozumiesz, dlaczego R może być powolny, wykorzystamy tę wiedzę do stopniowego ulepszania implementacji SMA, którą pokazaliśmy wcześniej, ale zanim to zrobimy, musimy nauczyć się mierzyć wydajność naszego kodu i na tym skupiamy się w następnej sekcji.

str. 18

Pomiar poprzez profilowanie i analizę porównawczą

Istnieje powszechne powiedzenie, które mówi, że nie można zmienić tego, czego nie można zmierzyć. Nawet jeśli technicznie możesz zmienić wydajność swojego kodu w R, na pewno nie będziesz w stanie wiedzieć, czy zmiana jest tego warta, jeśli jej nie zmierzysz. W tej sekcji przedstawimy trzy narzędzia, których możesz użyć do pomiaru kodu: Rprof(), system.time i microbenchmark(). Pierwsze dwa są zawarte w R, a trzeci wymaga zainstalowania pakietu microbenchmark. Narzędzie Rprof() służy do profilowania kodu, natomiast system.time() i microbenchmark() służy do testowania kodu. \*Profilowanie oznacza, że mierzysz, ile czasu dana implementacja spędza na każdej jej części. \*Benchmarking oznacza, że porównujesz łączny czas potrzebny na wykonanie różnych implementacji, aby porównać je między sobą, bez względu na ich wewnętrzne części.

str. 18

Podstawy profilowania z Rprof()

Nawet doświadczonym programistom trudno jest zidentyfikować wąskie gardła w swoim kodzie. Jeśli nie masz sporego doświadczenia i dobrego wyczucia, które części twojego kodu spowalniają jego wykonanie, prawdopodobnie lepiej będzie profilować swój kod, zanim zaczniesz go optymalizować. Dopiero po zidentyfikowaniu najważniejszych wąskich gardeł możesz podjąć próbę ich wyeliminowania. Trudno jest podać ogólne porady dotyczące poprawy wydajności, ponieważ każda implementacja jest zupełnie inna. Funkcja RProf() jest wbudowanym narzędziem do profilowania wykonywania funkcji języka R. W regularnych odstępach czasu profiler zatrzymuje interpreter, zapisuje bieżący stos wywołań funkcji i zapisuje informacje do pliku. Następnie możemy spojrzeć na podsumowania takich informacji, aby dowiedzieć się, gdzie nasza implementacja spędza najwięcej czasu. Pamiętaj, że wyniki z Rprof() są stochastyczne. Za każdym razem, gdy go użyjemy, wyniki będą nieco inne, w zależności od wielu rzeczy specyficznych dla twojego systemu, na które R nie ma wpływu. Dlatego wyniki, które otrzymujemy z Rprof(), są szacunkowe i mogą się różnić w ramach przykładowej realizacji. Aby użyć tej funkcji, po prostu wywołujemy ją bez parametrów, zanim wywołamy kod, który chcemy zmierzyć, a następnie wywołujemy ją ponownie, tym razem wysyłając parametr NULL. Wyniki są zapisywane w pliku na dysku twardym i można je wywołać za pomocą wywołania funkcji summaryRprof().

W tym konkretnym przypadku zwróć uwagę, że wysłaliśmy pierwsze 10000 elementów. Gdybyśmy wysłali niewielką ilość danych, funkcja sma\_slwo\_1() zakończyłaby się tak szybko, że nie mielibyśmy żadnego sensownego wyniku (pamiętaj, że Rprof() mierzy w odstępach czasu). Pokazane tutaj wyniki są również obcięte, ponieważ rzeczywiste wyniki są znacznie większe, ponieważ pokazują wiele wywołań funkcji, których użył nasz kod. Pozostawiliśmy pięć najlepszych wyników dla każdej tabeli. Obie tabele zawierają te same informacje. Różnica polega na tym, że tabela $by.self (pierwsza) jest uporządkowana według self, a tabela $by.total (druga) według total; self wskazuje, ile czasu zajęło wywołanie funkcji bez uwzględnienia wywołań funkcji potomnych, podczas gdy total informacje obejmują wywołania funkcji potomnych. Oznacza to, że dane muszą sumować się, podczas gdy dane zagregowane sumują się zwykle do znacznie większe niż 100:

Rprof()

sma\_1 <- sma\_slow\_1 (period, symbol, data\_original[1:10000, ])

Prof.(NULL)

summaryRprof()

#> $ by.self

#> self.time self.pct total.time total.pct

#> "rbind" 1,06 10,84 6,16 62,99

#> "struktura" 0,82 8,38 0,94 9,61

#> "data.frame" 0,68 6,95 4,32 44,17

#> "[.data.frame" 0,54 5,52 1,76 18,00

#> "sma\_slow\_1" 0,48 4,91 9,78 100,00

#> (Obcięte wyjście)

#>

#> $ by.total

#> total.time total.pct self.time self.pct

#> "sma\_slow\_1" 9,78 100,00 0,48 4,91

#> "rbind" 6,16 62,99 1,06 10,84

#> "data.frame" 4,32 44,17 0,68 6,95

#> "[" 1,88 19,22 0,20 2,04

#> "as.data.frame" 1,86 19,02 0,10 1,02

#> (Obcięte wyjście)

#>

#> $ sample.interval

#> [1] 0,02

#>

#> $ sampling.time

#> [1] 9.78

Jak widać w wynikach, pierwsza kolumna wskazuje wywołanie funkcji na stosie, a liczby wskazują, ile czasu zostało spędzone na określonym wywołaniu funkcji, w kategoriach bezwzględnych (time) lub względnych (pct). Zwykle będziesz chciał skupić się na najwyższych wartościach w kolumnie self.pct tabeli $by.self, ponieważ pokazują one funkcje, które same zajmują najwięcej czasu. W tym konkretnym przypadku rbind,structure, i data.frame są funkcjami, które zajmują najwięcej czasu. Na koniec powinieneś wiedzieć, że niektóre nazwy znalezione w stosie wywołań funkcji mogą być bardzo tajemnicze i czasami będziesz miał trudności ze znalezieniem odnośników lub dokumentacji do nich. Dzieje się tak, ponieważ są to prawdopodobnie wewnętrzne implementacje języka R, które nie są przeznaczone do bezpośredniego użytku przez użytkowników języka R. Sugeruję, aby po prostu spróbować naprawić te wywołania funkcji, które rozpoznajesz, chyba że masz do czynienia z sytuacjami, w których wysoce zoptymalizowany kod jest absolutnym wymogiem, ale w takim przypadku lepiej byłoby przeczytać specjalistyczną książkę na ten temat.

str. 20

Benchmarking ręczny za pomocą system.time ()

Teraz przyjrzymy się, jak przetestować Twój kod. Jeśli szukasz prostego pomiaru czasu realizacji, to syste.time to dobry wybór. Po prostu wywołujesz w nim funkcję, która wydrukuje dla Ciebie następujące trzy miary czasu:.

* user : Jest to czas user, na który powinniśmy zwrócić większą uwagę, ponieważ mierzy on czas procesora używany przez R do wykonania kodu
* system : Czas system jest miarą czasu spędzonego przez system, aby móc wykonać funkcję
* elapsed : Czas elapsed to całkowity czas potrzebny na wykonanie kodu, nawet jeśli został spowolniony z powodu innych procesów systemowych

Czasami czas elapsed jest dłuższy niż suma czasu user i czasu system, ponieważ procesor wykonuje wiele zadań jednocześnie z innymi procesami lub musi czekać na udostępnienie zasobów, takich jak pliki i połączenia sieciowe. W innych przypadkach czas, który upłynął, jest krótszy niż suma czasu i czasu. Może się tak zdarzyć, gdy do wykonania wyrażenia jest używanych wiele wątków lub procesorów. Na przykład zadanie, które zajmuje 10 sekund czasu użytkownika, może zostać wykonane w ciągu 5 sekund, jeśli obciążenie dzielą dwa procesory. Najczęściej jednak interesuje nas całkowity czas, jaki upłynął do wykonania danego wyrażenia. Gdy wyrażenie jest wykonywane w pojedynczym wątku (wartość domyślna dla języka R), upływający czas jest zwykle bardzo zbliżony do sumy czasu i czasu. Jeśli tak nie jest, albo wyrażenie spędziło czas, czekając na dostępność zasobów, albo w systemie było wiele innych procesów konkurujących o czas procesora. W każdym razie, jeśli podejrzewasz swoje pomiary, spróbuj mierzyć ten sam kod kilka razy, podczas gdy komputer nie zużywa zasobów na inne aplikacje. W tym konkretnym przypadku widzimy, że wykonanie zajęło około 9 sekund, aby zakończyć, co w przybliżeniu odpowiada temu samemu czasowi, jaki zajęło wykonanie go, mierząc Rprof() w poprzedniej sekcji, jak widać w kolumnie total.time w sma\_slow\_1() dotyczącej obserwacji tabeli $by.total.

system.time(sma\_slow\_1 (period, symbol. data\_original[1:10000, ]))

#> user system elapsed

#> 9.251 0.015 9.27

Jeśli chcesz zmierzyć wiele funkcji, aby porównać ich czasy, będziesz musiał użyć funkcji system.time ()na każdej z nich, więc jest to trochę ręczny proces. Lepszą alternatywą jest funkcja microbenchmark.

str. 21

Automatyczna analiza porównawcza za pomocą microbenchmark ()

Jeśli zidentyfikowałeś funkcję, która jest wielokrotnie wywoływana w Twoim kodzie i wymaga przyspieszenia, możesz napisać dla niej kilka implementacji i użyć funkcji micrbenchmark() z pakietu microbenchmark, aby je porównać. Jego wyniki będą również zwykle bardziej wiarygodne, ponieważ domyślnie uruchamia każdą funkcję 100 razy, dzięki czemu jest w stanie wygenerować statystyki dotyczące jej wydajności. Aby użyć tej funkcji, po prostu zawiń ją wokół fragmentu kodu, który chcesz zmierzyć. Niektóre przydatne funkcje polegają na tym, że możesz wykonać zadanie, w ramach którego bardzo przydatne jest zmierzenie i wykorzystanie wyników za jednym razem; możesz także przekazywać różne wywołania funkcji oddzielone przecinkami, a to da ci wyniki dla każdego z nich. W ten sposób można jednocześnie automatycznie porównywać różne funkcje. Tutaj przypiszemy wyniki sma\_slow\_1() do sma\_1, tak jak poprzednio, ale od tego czasu jest opakowany w funkcję microbenchmark(), zostanie również zmierzony, a wyniki wydajności zostaną zapisane w ramce danych performance. Ten obiekt zawiera następujące kolumny: exprneval to ciąg znaków, który zawiera użyte wywołanie funkcji, to ile razy funkcja została wykonana (domyślnie jest to 100) oraz min,lq (pierwszy kwartyl),mean,median,uq (trzeci kwartyl) i statystyki max:

performance <- microbenchmark (

sma\_1 <- sma\_slow\_1(period, symbl, data),

unit = „us”

)

summary(performance)#median

#> [1] 81035.19

Jeśli chcesz zobaczyć pełną ramkę danych wydajności, po prostu ją wydrukuj. Tutaj pokazaliśmy tylko, że czas median potrzebny do wykonania wywołania funkcji sma\_slow\_1() wynosił 81, 035.19 mikrosekundy (czyli jednostkę określoną w parametrze unit=”us”). Domyślnie zajęłoby to milisekundy zamiast mikrosekund, ale chcemy zapewnić te same jednostki dla wszystkich porównań, które wykonujemy i mikrosekund to lepsza opcja. Będziemy nadal dodawać rekordy do poniższej tabeli. Każdy wiersz będzie zawierał identyfikator implementacji, medianę w mikrosekundach potrzebnych do wykonania takiej funkcji, wskazanie najszybszej jak dotąd implementacji oraz procent w porównaniu z najszybszą, jaką mamy do tej pory. W tym konkretnym przypadku, ponieważ jest to jedyne, które zrobiliśmy, jest to oczywiście najszybsze, a także w 100% najlepsze, które samo w sobie jest:



[Najszybsze wdrożenie, w mikrosekundach, mediana% od najszybszego]

Celem pozostałej części jest rozszerzenie tej tabeli, aby zapewnić precyzyjne pomiary tego, jak wiele ulepszeń wydajności wprowadzamy w miarę ulepszania naszej implementacji SMA.

str. 23

Łatwe osiąganie wysokich korzyści - poprawa kosztów

W tej sekcji pokażemy, jak można radykalnie poprawić wydajność języka R bez uciekania się do zaawansowanych technik, takich jak delegowanie do innych języków programowania lub wdrażanie równoległości. Techniki te zostaną pokazane w dalszych sekcjach.

**Korzystanie z prostej struktury danych dla zadania**

Wielu użytkowników języka R zgodziłoby się, że ramka danych jako struktura danych jest podstawowym narzędziem do analizy danych. Zapewnia intuicyjny sposób reprezentowania typowego ustrukturyzowanego zbioru danych z wierszami i kolumnami reprezentującymi odpowiednio obserwacje i zmienne, ale zapewnia większą elastyczność niż macierz, umożliwiając zmienne różnych typów (takie jak zmienne znakowe i numeryczne w jednej strukturze). Ponadto, gdy ramki danych zawierają tylko zmienne numeryczne, podstawowe operacje macierzowe można wygodnie zastosować do nich bez konieczności jawnego wymuszania. Ta wygoda wiąże się jednak z kosztami wydajności, o których ludzie często nie wspominają. Tutaj unikamy powtarzania wyników Rprof(), które otrzymaliśmy z profilowania funkcji sma\_slow)1(). Jeśli jednak spojrzysz na nie, zobaczysz ,że rbind() i datat.frame() należą one do funkcji, które zajmowały najwięcej czasu. To jest właśnie wspomniany wcześniej koszt wydajności. Jeśli chcesz, aby implementacje były szybsze, unikanie ramek danych może być dobrym początkiem. Ramki danych mogą być doskonałym narzędziem do analizy danych, ale nie podczas pisania szybkiego kodu. Jak widać w sma\_slow\_2(), kod jest praktycznie taki sam jak sma\_slow\_1(), z wyjątkiem tego, że obiekt period\_prices() nie jest już ramką danych. Zamiast tego stał się wektorem, który jest rozszerzany funkcją c() zamiast funkcji rbind(). Zwróć uwagę, że wciąż dynamicznie zwiększamy rozmiar obiektu podczas wywoływania funkcji c(), czego nie powinieneś robić dla wykonującej ody, ale zrobimy to krok po kroku:

sma\_slow\_2 <- funtion(period, symbol, dat) {

result <- datat.frame(sma = numeric () )

for(end i 1:nrow(data)) {

position <- end

sma <- NA

n\_acumulated <- 0

period\_prices <- NULL

if(data[end, „symbol”] = = symbol) {

while(n\_accumulated < period & position >= 1) {

if (data[position, „symbol”] = = symbol) {

period\_prices <- c(period\_prices,

data[position, „price\_us:])

n\_acumulated <- n\_acumulated + 1

}

position <- position – 1

}

if (n\_acumulated = = period) {

sma <- 0

for (price in period\_prices) {

sma <- sma + price

}

sma <- sma / period

} else {

sma <- NA

}

result <- rbind(result, data.frame(sma=sma))

}

}

return(result)

}

W tym przypadku czas jego wykonania mierzymy tak jak wcześniej, ale wykonujemy też bardzo ważną weryfikację, która jest często pomijana. Weryfikujemy, że wartości, które otrzymujemy z sma\_slow\_1(), są takie same, jak te, które otrzymujemy z sma\_slow\_2() . Nie byłoby poprawnym porównaniem, gdybyśmy mierzyli implementacje, które robią różne rzeczy. Wykonanie sprawdzenia jest również przydatne, aby zwiększyć naszą pewność, że każda wprowadzana przez nas zmiana nie powoduje nieoczekiwanego zachowania. Jak widać, wszystkie wartości są takie same, więc możemy kontynuować z pewnością:

performance <- microbenchmark (

sma\_2 <- sma\_slow\_2(period, symbol, data),

unit = „us”

)

all(sma\_1$sma ==sma\_2$sma, na.rm = TRUE\_

>#> TRUE

summary(performance)$median

#> [1] 33031.7785

Rejestrujemy nasze wyniki w naszej tabeli i zdajemy sobie sprawę, że usunięcie tej struktury ramki danych pozwoliło nam usunąć dwie trzecie czasu wykonywania. To całkiem nieźle jak na tak łatwą zmianę, prawda? Ponieważ nasz podstawowy przypadek (najszybsza implementacja, jaką mamy do tej pory) to sma\_slow\_2(), widzimy, że sma\_slow\_1() wykonanie zajmie około 145% więcej czasu:



Teraz, gdy zdajemy sobie sprawę, jaki wpływ mogą mieć niepotrzebne ramki danych na działanie naszego kodu, przystępujemy do usuwania również drugiej ramki danych, której używaliśmy dla obiektu result. Zastępujemy go również wektorem i używamy funkcji c(), aby do niego dołączyć. Ten sam problem dynamicznej ekspansji, o którym mowa wcześniej, pojawia się również tutaj. Jak widać, wszystko inne jest takie samo. Przechodzimy do testu porównawczego, tak jak robiliśmy to wcześniej, i sprawdzamy również, czy otrzymane wyniki są takie same. Ostrożny czytelnik mógł zauważyć, że poprzednia kontrola została przeprowadzona z operatorem równości, podczas gdy ta została przeprowadzona z operatorem nierówności. W rzeczywistości, sprawdzając liczby rzeczywiste, lepiej jest sprawdzić, czy są one wystarczająco zbliżone, a nie dokładnie takie same. Jeśli sprawdziłeś identyczne liczby, możesz otrzymać wynik FALSE, ponieważ jedna z liczb ma różnicę 0.000000001, która w naszym przypadku nie jest istotna. Dlatego ustalamy, co jest istotnym sprawdzianem dla naszego konkretnego przypadku użycia i sprawdzamy, czy każda para liczb ma różnicę nie większą niż ten próg, tak jak robimy tutaj, z naszym progiem 0.001:

performance <- microbenchmark(

sma\_3 <- sma\_slow\_3 (period, symbol, data),

unit = „us”

)

all(sma\_1$sma- sma\_3 <= 0.001,na.rm = TRUE)

#> TRUE

summary(performance)$mediaan

#> [1] 19628.243

W tym przypadku średni czas potrzebny do wykonaniasma\_slow\_3() wyniósł 19, 628.243 mikrosekundy. Kontynuujemy i zapisujemy to w naszej tabeli i ponownie obliczamy procent od najlepszego, który jest sma\_slow\_3() w tym momencie. Zauważ, że jesteśmy w stanie usunąć prawie połowę czasu z już ulepszonej funkcji sma\_slow\_2(), a użycie oryginalnej funkcji sma\_slow\_1() zajmie 312% więcej czasu niż najnowsza. Zaskakujące może być to, jak duży wzrost wydajności można uzyskać, używając prostszej struktury danych

str. 26

Wektoryzacja tak bardzo, jak to możliwe

Wektoryzacja oznacza usunięcie ręcznego mechanizmu zapętlania na korzyść operacji zoptymalizowanej do zrobienia tego samego bez potrzeby jawnej pętli. Jest to bardzo pomocne, ponieważ pomaga uniknąć narzutu związanego z jawnymi pętlami w R. Wektoryzacja jest podstawowym narzędziem w języku R i należy przyzwyczaić się do programowania przy użyciu go zamiast używania jawnych pętli, gdy tylko jest to możliwe, bez czekania, aż nadejdzie etap wydajności do gry. Kiedy zrozumiesz, jak to działa, przyjdzie naturalnie. Istnieją różne sposoby wektoryzacji operacji. Na przykład, jeśli chcesz wykonać mnożenie macierzy wektorów, zamiast iterować po elementach wektora i macierzy, mnożąc odpowiednie współczynniki i dodając je razem, jak to zwykle robi się w innych językach programowania, możesz po prostu zrobić coś w rodzaju A %\*% A aby wykonać wszystkie te operacje w sposób zwektoryzowany w R. Wektoryzacja zapewnia bardziej wyrazisty kod, który jest łatwiejszy do zrozumienia i wydajniejszy, dlatego zawsze należy próbować go używać. Innym sposobem wektoryzacji jest użycie rodziny funkcji apply() R (na przykład, lapply() ,sapply() i tak dalej). Spowoduje to wygenerowanie prostszego kodu niż jawne pętle, a także przyspieszy implementację. W rzeczywistości funkcja apply() jest przypadkiem szczególnym, ponieważ nie jest tak zoptymalizowana, jak inne funkcje z jej rodziny, więc wzrost wydajności nie będzie tak duży, jak w przypadku innych funkcji, ale przejrzystość kodu rzeczywiście wzrośnie. Innym sposobem wektoryzacji kodu jest zastąpienie pętli wbudowanymi funkcjami języka R i tak właśnie będzie w następnej modyfikacji. W trzecim if w kodzie, tym po pętla while się skończyła, istnieje pętla for, która dodaje elementy, które mamy w wektorze period\_prices, a następnie jest dzielona przez wektor period, aby uzyskać średnią. Możemy po prostu użyć funkcji mean() zamiast korzystać z takiej pętli i to właśnie robimy. Teraz, kiedy czytasz tę część kodu, łatwo odczytuje się, jakby liczba skumulowanych cen była równa okresowi, co sprawia, że ​​SMA jest równe średniej skumulowanych cen. Znacznie łatwiej jest zrozumieć kod niż użycie pętli:

sma\_slow\_4 <- function(period, symbol, data) {

resultl <- NULL

for(end in 1:nrow(data) ) {

position <- end

sma <- NA

n\_accumulated <- 0

period\_proces <- NULL

if (data[end, „symbol”] == symbol) {

while(n\_acumulated < eriod & position >- 1) {

if (data[position, „symbol”] = = symbol) {

priod\_prices <- c(period\_prices,

data[position, „price\_isd”])

n\_acumulated <- n\_accumulated + 1

}

position <- position -1

}

if (n\_aumulated = = period) {

sma <- mean(period\_prices)

} else {

sma <- NA

result <- c(result, sma)

}

}

return(result)

}

Ponownie porównujemy i sprawdzamy poprawność. Jednak w tym przypadku okazuje się, że średni czas to 20,825.879 mikrosekundy, czyli więcej niż bieżące minimum od sma\_slow\_3(). Czy kod wektorowy nie powinien być szybszy? Odpowiedź jest taka, że ​​przez większość czasu tak jest, ale w takich sytuacjach funkcja mean() ma narzut, ponieważ musi sprawdzić, z jakim typem obiektu ma do czynienia, przed użyciem jej do jakichkolwiek operacji, co może spowalniają implementację. Kiedy używaliśmy jawnej pętli, sumy i dzielenie były znacznie niższe, ponieważ można je było zastosować do znacznie mniejszego zestawu obiektów. Dlatego, jak widać w poniższej tabeli, sma\_slow\_4() zajmuje o 6% więcej czasu niż sma\_slow\_3(). To niewiele, a ponieważ wolę kod ekspresyjny, zachowam zmianę:

performance <- mirobenchmark (

sma\_4 <- sma\_slow\_4 (period, symbol ,data) ,

unit =”us”

)

all(sma\_1$sma – sma\_4 <= 0.001 , na.rm = TRUE)

#> TRUE

 summary(performance)$median

#> [1] 20825.8790

Spójrz na poniższą tabelę:



Jeśli chcesz porównać narzut funkcji mean() z narzutem wynikającym z innych sposobów wykonywania tych samych obliczeń, spójrz na poniższy punkt odniesienia. Funkcja .Internal(mean(x)) omija mechanizm wysyłania metod, które pokazaliśmy w poprzednim rozdziale i przeskakuje bezpośrednio do implementacji funkcji mean() w C, jak pokazano w poniższym fragmencie kodu:

x <- sample(100)

performance <- microbenchmark (

mean(x),

sum(x) / length(x),

.Internal(mean(x)),

Times = 1e+05

)

performance

#> Jednostka: nanosekundy

#> wyr min lq średnia mediana uq max neval

#> średnia (x) 1518 1797 2238,2607 1987 2230 2335285 1e + 05

#> suma (x) / długość (x) 291345 750.2324 403488 27016544 1e + 05

#>. Wewnętrzna (średnia (x)) 138153 187.0588 160176 34513 1e + 05

str. 29

Usunięcie niepotrzebnej logiki

Są chwile, kiedy prosta logika pokazuje nam, że są części naszych implementacji, które są niepotrzebne. W tym konkretnym przypadku kumulacji period\_prices można uniknąć, ustawiając sma na 0 początkowo zamiast NA i dodając do niej każdą cenę. Jednak robiąc to, tracimy kontrolę nad liczbą elementów w wektorze, więc funkcja mean() nie ma już sensu i przystępujemy do podzielenia sumy przez period tak, jak robiliśmy to wcześniej:

sma\_slow\_5 <- function(period, symbol, data) {

result <- NULL

for(end in 1:nrow(data)) {

position <- end

sma <- 0

n\_acumulated <- 0

if (data[end , „symbol”] = = symbol) {

while(n\_acumulated < period & position >- 1) {

if (data[position, „symbol”] = = symbol) {

sma <- sma + data[position, „price\_usd”]

n\_accumulated <- n\_accumulated + 1

}

position <- position -1

}

if (n\_acumulated == period) {

sma <- sma / period

} else {

sma <- NA

}

result <- c(result, sma)

}

}

return(reuslt)

}

Ponownie porównujemy i sprawdzamy poprawność, jak pokazano w poniższym fragmencie kodu:

performance <- microbenchmark(

sma\_5 <- sma\_slow\_5(period, symbol, data),

unit = „us”

)

all(sma\_1$sma – sma\_5 <= 0.001, na.rm = TRUE)

#> TRUE

summary(performance)$median

#> [1] 16682.68

W tym przypadku nasz średni czas wyniósł 16682.68 mikrosekundy, co sprawia, że jak dotąd jest to najszybsza implementacja. Ponownie zwróć uwagę, że bardzo prosta zmiana spowodowała redukcję o około 17% w porównaniu z wcześniejszą najszybszą implementacją:



str. 30

Przenoszenie sprawdzeń z procesów iteracyjnych

Załóżmy, że utknęliśmy w naszym procesie optymalizacji i nie wiemy, co powinniśmy teraz zmienić. Co powinniśmy zrobić? Cóż, jak wspomnieliśmy wcześniej, powinniśmy sprofilować nasz kod, aby poznać nasze obecne wąskie gardła i to właśnie robimy tutaj. Używamy tej funkcji Rprof() ponownie do profilowania naszej implementacji sma\_slow\_5(). Wyniki pokazują, że funkcje [.data.frame i [ są naszymi największymi wąskimi gardłami i chociaż ich nazwy są nieco tajemnicze, możemy się domyślić, że są one związane z podziałem ramek danych (którymi są). Oznacza to, że naszym obecnie najważniejszym wąskim gardłem jest sprawdzenie, czy znajdujemy się w obserwacji, która odpowiada używanej przez nas obserwacji, i wykonujemy takie sprawdzenia w różnych miejscach w naszym kodzie:

Rpof()

sma\_5 <- sma\_slow\_5(period, symbol, data\_original[1:10000, ] )

Rprof (NULL}

summaryRprof()

#> $ by.self

#> self.time self.pct total.time total.pct

#> „[.data.frame” 0,54 26,21 1,24 60,19

#> "[" 0,22 10,68 1,34 65,05

#> „NextMethod” 0,20 9,71 0,20 9,71

#> "sma\_slow\_5" 0,14 6,80 2,06 100,00

#> "Ops.factor" 0,12 5,83 0,52 25,24

#> (Obcięte wyjście)

#>

#> $ by.total

#> total.time total.pct self.time self.pct

#> "sma\_slow\_5" 2,06 100,00 0,14 6,80

#> "[" 1,34 65,05 0,22 10,68

#> "[.data.frame" 1,24 60,19 0,54 26,21

#> „Ops.factor” 0,52 25,24 0,12 5,83

#> „NextMethod” 0,20 9,71 0,20 9,71

#> (Obcięte wyjście)

Teraz, gdy znamy nasze obecne największe wąskie gardło, możemy je usunąć, unikając sprawdzania, czy bieżąca obserwacja odpowiada symbol, którą otrzymujemy jako parametr. Aby to osiągnąć, po prostu wprowadzamy filtr na początku funkcji, który przechowuje tylko obserwacje zawierające prawidłowy symbol. Zwróć uwagę, że ten prosty filtr pozwala nam usunąć dwie kontrole, które wykonywaliśmy wcześniej, ponieważ jesteśmy pewni, że wszystkie obserwacje mają poprawny symbol. Zmniejsza to dwa poziomy wcięć w naszym kodzie, ponieważ te sprawdzenia zostały zagnieżdżone. To wspaniałe uczucie, prawda? Teraz wydaje się, że mamy bardzo prostą implementację, która intuicyjnie będzie działać znacznie lepiej. Aby to zweryfikować, przechodzimy do benchmarku i sprawdzamy poprawność, jak wcześniej:

performance <- microbenchmark {

sma+6 <- sma\_slow\_6(perid, symbol, data),

unit =”us”

)

all(sma\_1$sma – sma\_6 <- 0.001, na.rm = TRUE)

#> TRUE

summary(performance)$median

#> [1] 2991.5720

Potwierdza się również nasza intuicja; nasz średni czas dla sma\_slow\_6() to 2, 991.57. To jest tylko 17% w porównaniu z wcześniejszą najszybszą implementacją, którą mieliśmy dla sma\_slow\_5() , i zajmuje to tylko 3% czasu, jaki zajmowała nasza pierwsza implementacja. Czy to jest niesamowite, czy co? Spójrz na poniższą tabelę:



str. 32

Jeśli możesz, w ogóle unikaj iteracji

W poprzedniej sekcji zdaliśmy sobie sprawę, jak duży wpływ na wydajność naszej implementacji może mieć niepotrzebny narzut w ramach iteracji. A co by było, gdybyśmy w ogóle mogli uniknąć iteracji? Teraz byłoby lepiej, prawda? Cóż, jak wspomnieliśmy wcześniej, robienie tego jest osiągalne dzięki wektoryzacji. W takim przypadku usuniemy pętlę while i zastąpimy ją wektoryzowaną średnią nad pozycjami start i end , gdzie nadal jest definiowana tak, jak dotychczas, a start jest definiowana jako pozycja end minus period otrzymana jako parametr plus jeden. Gwarantuje to, że otrzymamy dokładną liczbę cen, których potrzebujemy, i możemy utworzyć przedział start:end, który będzie pobierał określony podzbiór, którego potrzebujemy z data, abyśmy mogli zastosować do niego funkcję mean():

sma\_slow\_7 <- funtion(period, symbol, data) {

data <- data[data$symbol = = symbol, ]

result <- NULL

for (end in 1:nrow(data)) {

start <- end – period + 1

if (start >= 1) {

sma <- mean(data[start:end, „price\_usd”])

} else {

sma <-NA

}

result<- c(result, sma)

}

return(result)

}

Zauważ, że ta zmiana nie byłaby możliwa, gdybyśmy nie przefiltrowali danych u góry funkcji, ponieważ mielibyśmy obserwacje, które odpowiadają różnym symbolom zmieszanym między sobą, a nasz przedział start:end wybrałby obserwacje zawierające inne symbole. To pokazuje, że czasami optymalizacje zależą od siebie nawzajem, a jednej nie można zastosować bez zastosowania poprzedniej, a te relacje często występują przypadkowo. Jak zawsze wykonujemy testy porównawcze i sprawdzamy poprawność, jak pokazano w poniższym fragmencie kodu:

performance <- microbenchmark (

sma\_7 <- sma\_slow\_7(period, symbol, data),

unit= „us”

all(sma\_1$sma – sma\_7 <= 0.001, na.rm = TRUE)

#> TRUE

summary(performance\_$median

#> [1] 910.793

Mediana czasu to teraz 910. 793 mikrosekundy. Spodziewaliśmy się tego, ponieważ wiemy, że usunięcie jawnych pętli może spowodować znaczną poprawę wydajności. W tym przypadku udało nam się skrócić do nieco poniżej jednej trzeciej czasu w porównaniu z poprzednio najszybszym wdrożeniem. Zauważ, że mamy teraz do czynienia z setkami mikrosekund zamiast tysięcy mikrosekund. Oznacza to, że osiągnęliśmy poprawę wydajności o rzędy wielkości. Spójrz na poniższą tabelę:



str. 34

Efektywne korzystanie ze sposobu iteracji w języku R.

W tym momencie pozostaje nam pojedyncza pętla for, którą chcielibyśmy usunąć. Jednak jest w tym trochę logiki, która przeszkadza. W tym miejscu funkcja lapply() jest przydatna. Funkcja ta otrzymuje listę obiektów, które zostaną wysłane do funkcji podanej jako drugi argument i zwróci wyniki takich wywołań funkcji na liście. Dodatkową zaletą tej funkcji jest to, że zajmuje się ona wstępną alokacją pamięci za nas, co jest bardzo wydajnym sposobem na skrócenie czasu wykonywania w R. W tym przypadku zamykamy logikę wewnątrz naszej pętli for w oddzielnej funkcji o nazwie sma\_from\_position\_1() i używamy jej w ramach naszego wywołania funkcji lapply() . Nasza funkcja sma\_from\_position\_1() otrzymuje obiekty end, period i data , z którymi pracowaliśmy, i zachowują one to samo znaczenie i wykonują te same obliczenia średniej wektorowej, które robiliśmy wcześniej. Jednak zamiast używać jawnego warunku if … else , używa funkcji ifelse(), która przyjmuje warunek do sprawdzenia jako pierwszy argument, pożądany wynik w przypadku spełnienia warunku jako drugi argument, a pożądany wynik w przypadku, gdy warunek nie zostanie spełniony jako trzeci argument. W naszym przypadku są to odpowiednio start >= 1, mean(data[start:end], price\_usd i NA , odpowiednio. Wynik, który otrzymujemy z wywołań funkcji sma\_from\_position\_1(), jest nielistowany w jednym wektorze, dzięki czemu otrzymujemy wynik w postaci wektora zamiast listy, który z kolei jest zwracany przez. Zwróć uwagę na zmianę nazwy? Na tym etapie implementację tę można uznać za skuteczną. Hurra! Spójrz na następujący fragment kodu:

sma\_efficient\_1 <- function(period , symbol, data) {

data <- data[data$symbol = = symbol, ]

return(unlsit(lapply(1;nrow(data),

sms\_from\_position\_1,

period, data)))

}

sma\_from\_position\_1 <- function(end, period, data) {

start <- end – period + 1

return(ifelse(start >= 1,

mean(data[start:end, „price\_usd”]), NA))

}

Na wypadek, gdybyś nie pamiętał mechaniki funkcji lapply() i był trochę zdezorientowany sposobem jej użycia, przypomnę, że weźmie każdy element z listy podanej jako pierwszy argument, i podać je jako pierwszy argument funkcji podanej w drugim argumencie. Jeśli wspomniana funkcja wymaga większej liczby parametrów, można je również przekazać po dostarczeniu obiektu funkcji do funkcji lapply(), co ma miejsce w przypadku argumentów period i data, które widzisz na końcu. Ponownie wykonaj test porównawczy i sprawdź poprawność, jak pokazano w poniższym fragmencie kodu:

performance <- micronbenchmark(

sma\_8 <- sma\_efficient\_1(period, symbol, data),

unit = „us”

)

all(sma\_1$sma – sma\_8 <= 0.001, na.rm = TRUE)

#> TRUE

summary(performance)$median

#> [1] 1137.704

Tym razem nasz średni czas to 1,137.704 mikrosekundy. To więcej niż nasza dotychczas najszybsza implementacja. Co się stało? Jeśli chcesz poznać szczegóły, powinieneś sprofilować funkcję, ale w istocie problem polega na tym, że dodajemy wywołanie funkcji, które jest wykonywane wiele razy (sma\_from\_position\_1()), a wywołania funkcji mogą być drogie, a także dodawanie transformacji z list do wektora, którego nie robiliśmy wcześniej (unlist()). Jednak wolimy przejść dalej z wersją z powodów, które zostaną wyjaśnione w dalszej części. Istnieje wiele innych funkcji wektoryzowanych w R, które mogą pomóc przyspieszyć twój kod. Niektóre przykłady to which(), where(), any(), all(), cumsum() i cumprod(). Podczas pracy z macierzami możesz używać rowSums(), colSums() lower.tri() , upper.tri() i innych, a podczas pracy z kombinacjami możesz używać combin(). Jest ich o wiele więcej, a kiedy mamy do czynienia z czymś, co wydaje się być wektoryzowane, istnieje szansa, że ​​istnieje już funkcja do tego.

str. 36

Unikanie wysyłania struktur danych z narzutami

Wiemy, że w miarę możliwości należy unikać operowania na ciężkich strukturach danych, takich jak ramki danych, a tutaj wydaje się, że nadal jest to możliwe. A co, jeśli zamiast przekazywać naszą ramkę danych, wyodrębnimy interesującą nas zmienną price\_usd i po prostu jej użyjemy? To wydaje się obiecujące. Aby to osiągnąć, w górnej części funkcji nie tylko filtrujemy obserwacje, które zawierają symbol nam potrzebne, ale także wyodrębniamy zmienną price\_usd w tym miejscu. Teraz możemy wysłać tę strukturę danych o niższym narzucie do naszej nieco zmodyfikowanej funkcji sma\_from\_position\_2(). Jest po prostu zmodyfikowany, aby działał z tym wektorem zamiast z pełną ramką danych:

sma\_efficient\_2 <- function(period, symbol, data) {

data <- data[datat$symbol = = symbol, „price\_usd”]

return(unlist(lapply(1:length(data),

sma\_from\_position\_2,

period, data)))

}

sma\_from\_position\_2 <- function(end, period, data) {

start <- end – period +1

return(ifelse(start >= 1, sum(data[start:end]) / period, NA))

}

Ponownie wykonaj test porównawczy i sprawdź poprawność, jak pokazano w poniższym fragmencie kodu:

performance <- microbenchmark (

sma\_9 <- sma\_efficient\_2(period, symbol, data),

unit = „us”

)

all(sma\_1$sma – sma-9 <= 0.001 , na.rm = TRUE)

#> TRUE

summary(performance)$median

#> [1] 238.2425

Tym razem nasz średni czas to 238.2425 mikrosekundy. To duża zmiana. W rzeczywistości jest to największa poprawa wydajności, jaką byliśmy w stanie osiągnąć, biorąc pod uwagę ilość wymaganych zmian w porównaniu z poprzednio najszybszą implementacją. Czy zdajesz sobie sprawę, jak drastyczna była poprawa wydajności? Wykonanie naszej pierwszej implementacji zajmuje około 33 900% więcej czasu. I odwrotnie, nasze wdrożenie sum\_efficient\_2() zajmuje tylko około 0,2% czasu, jaki zajęło nasze wdrożenie. Czy spodziewaliście się tak dużej redukcji czasu, pisząc tylko lepszy kod R, kiedy zaczynaliśmy? Załóżmy, że jesteśmy bardzo wybredni i chcemy dalej poprawiać wydajność. Co powinniśmy zrobić? Cóż, sprofilujmy ponownie nasz kod, aby się dowiedzieć. Jak widać tutaj, liczba wywołań funkcji jest zredukowana do jednego w tabeli $by.self i tylko do pięciu w tabeli $by.total. Niestety, te wyniki nie pokazują nam żadnej drogi, którą możemy dalej poprawić wydajność, ponieważ wszystkie pokazane funkcje są już wysoce zoptymalizowane. Jedyne, co możesz spróbować, to zastąpić funkcję mean() jedną z szybszych alternatyw pokazanych wcześniej, ale nie zrobimy tego w tym przypadku, ponieważ efekt tego został już pokazany wcześniej:

Rprof()

sma\_9 <- sma\_efficient\_2(periodm symbol, data\_original[1:10000, ])

Rprof(NULL)

summaryRptof()

#> $ by.self

#> self.time self.pct total.time total.pct

#> "ifelse" 0,02 100 0,02 100

#>

#> $ by.total

#> total.time total.pct self.time self.pct

#> "ifelse" 0,02 100 0,02 100

#> „ZABAWA” 0,02 100 0,00 0

#> "lapply" 0,02 100 0,00 0

#> "sma\_efficient\_2" 0,02 100 0,00 0

#> "unlist" 0,02 100 0,00 0

Aby jeszcze bardziej skrócić czas wykonania naszej implementacji, będziemy musieli skorzystać z bardziej zaawansowanych technik, takich jak zrównoleglanie i delegowanie, które są przedmiotem kolejnych sekcji.

Zauważ, że w tym miejscu Rprof() przestanie być użyteczne przez większość czasu, ponieważ zaczniemy używać zaawansowanych narzędzi, poza R, aby nadal poprawiać wydajność, a takie narzędzia wymagają własnych technik profilowania i wiedzy, którą nie będziemy się zajmować

str. 38**<-TU**

Używanie równoległości do dzielenia i podbijania

Do tej pory poznaliśmy różne sposoby optymalizacji wydajności programów R uruchamianych szeregowo, czyli w jednym wątku. Nie wykorzystuje to wielu rdzeni procesora, które większość komputerów ma obecnie. Obliczenia równoległe pozwalają nam je wykorzystać, dzieląc nasze implementacje na wiele części, które są niezależnie wysyłane do tych procesorów, i mogą przyspieszyć działanie programów, gdy jeden wątek jest ważnym wąskim gardłem. Równoległe tworzenie aplikacji w świecie rzeczywistym może być bardzo trudnym zadaniem i wymaga głębokiej wiedzy na temat oprogramowania oraz sprzętu. Zakres możliwej równoległości zależy od konkretnego algorytmu, z którym pracujemy, i jest dostępnych wiele typów równoległości. Ponadto zrównoleglenie nie jest decyzją tak / nie; wymaga ciągłej skali. Po jednej stronie skali mamy żenująco równoległe zadania, w których nie ma zależności między równoległymi podzadaniami, co czyni je doskonałymi kandydatami do zrównoleglenia. Z drugiej strony mamy zadania, których w ogóle nie można zrównoleglać, ponieważ każdy krok zadania zależy od wyników poprzednich kroków. Większość algorytmów mieści się między tymi dwoma skrajnościami, a większość aplikacji równoległych w świecie rzeczywistym wykonuje niektóre zadania szeregowo, a inne równolegle. Niektóre zadania, które są stosunkowo łatwe do wykonania równolegle (niektóre z nich zostałyby sklasyfikowane jako zawstydzająco równoległe), to konwertowanie setek obrazów z kolorów na skalę szarości, dodawanie milionów liczb, wyszukiwanie siłowe i symulacje Monte Carlo. Wspólną właściwością wśród nich jest to, że każde podzadanie można wykonać niezależnie od pozostałych. Na przykład każdy obraz może być przetwarzany niezależnie lub możemy dodać różne podgrupy liczb, a następnie zsumować wyniki i tak dalej. W momencie, gdy wprowadzamy zależność od kolejności, następuje zrównoleglenie.

str. 39

Jak głęboka jest królicza nora równoległa?

W przypadku równoległości i algorytmu jest wiele decyzji, które należy podjąć. Przede wszystkim musimy zdecydować, które części algorytmu będą implementowane równolegle, a które seryjnie oraz jak zarządzać tymi częściami, aby poprawnie działały między sobą. Następnie musimy zdecydować, czy to jawnie, czy niejawnie, czy zrównoleglone części będą miały pamięć współdzieloną czy rozproszoną, czy będziemy wykonywać zrównoleglenie danych lub zadań, czy musimy wprowadzić jakiś rodzaj mechanizmu rozproszonego czy współbieżnego, a jeśli tak, to jaki protokół będzie być używane do ich koordynowania. Po podjęciu decyzji na wysokim poziomie musimy zająć się szczegółowymi decyzjami dotyczącymi liczby i architektury procesorów, których będziemy używać, a także ilości pamięci i uprawnień kontrolnych. Nie przejmuj się zbytnio koncepcjami wspomnianymi wcześniej; są przeznaczone do bardziej zaawansowanych zastosowań niż planowany poziom w tej książce. Podam tutaj bardzo ogólne i proste wyjaśnienia, aby upewnić się, że rozumiesz typ równoległości, który sami wdrożymy, ale możesz pominąć tę sekcję, jeśli chcesz. Systemy pamięci współdzielonej współużytkują obiekty przechowywane w pamięci w różnych procesach, co może być bardzo wydajne pod względem zasobów, ale także niebezpieczne, ponieważ jeden proces może modyfikować obiekt, który jest używany przez inny proces, nie wiedząc, że to się stało. Inną wadą takich systemów jest to, że nie skalują się dobrze. Mocniejszą, ale także bardziej złożoną alternatywą jest pamięć rozproszona, która tworzy kopie danych potrzebnych do różnych procesów, które mogą znajdować się w różnych systemach. To podejście można skalować do tysięcy procesorów, ale odbywa się kosztem złożonej koordynacji między procesami. Paralelizm danych występuje wtedy, gdy dane są podzielone na partycje, a każde zadanie jest wykonywane przy użyciu innej partycji. Te typy zrównoleglania pomagają skalować algorytm w miarę gromadzenia większej ilości danych, ponieważ możemy po prostu utworzyć więcej partycji. Należy zauważyć, że użycie równoległości danych niekoniecznie oznacza pamięć rozproszoną i na odwrót. Równoległość zadań występuje, gdy zadania są wysyłane do różnych procesorów w celu wykonania równoległego i mogą, ale nie muszą, działać na wierzchu

tych samych danych. Wadą obliczeń równoległych jest to, że ludzie uruchamiają kod na różnych maszynach, a jeśli piszesz oprogramowanie, które chcesz udostępnić innym, musisz uważać, aby implementacja była użyteczna nawet wtedy, gdy jest wykonywana na różnych konfiguracjach sprzętowych. Wszystkie wspomniane wcześniej decyzje wymagają do prawidłowego podjęcia głębokiej wiedzy technicznej, a jeśli wydają się skomplikowane, to dlatego, że tak naprawdę są. Wdrażanie równoległości może być dość złożoną czynnością, w zależności od poziomu kontroli, jaki chcesz mieć nad nią. Co najważniejsze, pamiętaj, że R jest językiem interpretowanym, więc wzrost szybkości wynikający z używania języków kompilowanych prawie zawsze będzie przekraczał przyrost prędkości wynikający z równoległego tworzenia pętli lub innych funkcji ukrywania pętli.

str. 40

Praktyczna równoległość z R

W tej sekcji pokażemy, jak wykorzystać wiele rdzeni w R. Pokażemy, jak wykonać pojedynczy system pamięci współdzielonej z podejściem wielu rdzeni. To najprostsza technika równoległa, jaką możesz zastosować. Wdrażanie programów równoległych w R stało się z czasem coraz łatwiejsze, ponieważ jest to temat bardzo interesujący, wiele osób zapewniło i nadal zapewnia lepsze sposoby osiągnięcia tego celu. Obecnie w CRAN jest ponad 70 pakietów, które zapewniają pewnego rodzaju funkcje zrównoleglania. Wybór odpowiedniego pakietu dla właściwego problemu lub po prostu świadomość, że istnieje wiele opcji, pozostaje wyzwaniem. W tym przypadku użyjemy pakietu parallel , który jest preinstalowany w ostatnich wersjach R. Inne bardzo popularne pakiety to doSNOW i foreach ale to naprawdę zależy od tego, jakiego rodzaju zrównoleglenie chcesz wykonać. Najbardziej powszechną techniką zrównoleglania w języku R jest użycie zrównoleglonych zamienników funkcji lapply(), sapply() i apply(). W przypadku pakietu parallel mamy odpowiednio dostępne funkcje parLapply(),parSapply() i parApply(). Fakt, że sygnatury wśród tych par funkcji są bardzo podobne, sprawia, że ​​bariera w stosowaniu tej formy zrównoleglenia jest bardzo mała, dlatego postanowiłem zaprezentować tę technikę. Implementacja techniki zrównoleglania, którą pokażemy, jest dość prosta i obejmuje następujące trzy główne kroki po załadowaniu pakietu:

1. Utwórz klaster z funkcją makeCluster()

2. Zastąp funkcję apply() jedną par\*pply()

3. Zatrzymaj klaster utworzony w pierwszym kroku

W naszym przypadku zastąpimy funkcję lapply() parLapply() w naszej implementacji sma\_efficient\_2(). Należy jednak unikać częstego błędu popełnianego przez ludzi, którzy dopiero rozpoczynają pracę z równoległością. Zwykle tworzą, a później niszczą klaster w ramach funkcji wywoływanej do wykonania zadania, zamiast odbierać klaster z zewnątrz i używać go wewnątrz. Stwarza to problemy z wydajnością, ponieważ klaster będzie potencjalnie uruchamiany wiele razy, a uruchomienie klastra z równoległością może wiązać się z dużym narzutem. Funkcją, która popełnia taki błąd, jest funkcja sma\_parallel\_inefficient(), jak następuje:

librat(parallel)

sma\_parallel\_inefficient <- function(period, symbol, data) {

data <- as.numeric(data[data$symbol = = symbol, „price\_usd”])

cluster <- makeCluster(detectCores())

result <- unlist(parLapply(

cluster , 1:length(data) , sma\_from\_position\_2, period, data))

stopCluster(cluster)

return(result)

}

Jak widać, sma\_parallel)inefficient() jest tylko sma\_efficient\_2() z dodaną logiką do tworzenia i usuwania klastra oraz zamiany lapply() za pomocą parLapply(). Tak naprawdę nie powinieneś używać tej funkcji, ale została umieszczona tutaj, aby pokazać, jak źle może to wpłynąć na wydajność, jeśli to zrobisz. Jak zawsze wykonujemy testy porównawcze i sprawdzamy poprawność, jak pokazano w poniższym fragmencie kodu:

performance <- microbenchmark (

sma\_10 <- sma\_parallel\_inefficient(period, symbol, data),

unit = „us”

)

all(sma\_1$sma – sma\_10 <- 0.001, na.rm =- TRUE)

#> TRUE

summary(performance)$median

#> [1] 1197329.3980

W tym przypadku nasza mediana czasu to 1,197,329.398 mikrosekundy, co nie powinno być zbyt zaskakujące, gdy wspomnimy, że wielokrotne tworzenie i niszczenie klastra może być dość nieefektywne. Teraz przystępujemy do usunięcia logiki, która tworzy i niszczy klaster z funkcji, a zamiast tego otrzymaj cluster jako parametr do sma\_parallel(). W tym wypadku, nasza implementacja wygląda tak samo jak poprzednio, z wyjątkiem użycia parLapply(). Fajnie jest móc osiągnąć coś tak złożonego jak zrównoleglenie za pomocą prostej zmiany, ale tak naprawdę jest to produkt uproszczenia naszego kodu do tego, co mamy teraz. Gdybyśmy spróbowali zrównoleglać naszą początkową implementację sma\_slow\_1(), byłoby to trudne. Spójrz na następujący fragment kodu:

sma\_parallel <- function(period, symbol, data, cluster) {

data <- as.numeric(data[data$symbol == symbol, „price\_usd”])

return(unlist(parLapply(

cluster, 1:length(data), sma\_from\_position\_2(), period, data )))

}

Ponownie wykonujemy testy porównawcze i sprawdzamy poprawność, jak pokazano w poniższym fragmencie kodu:

cluster <- makeCluster(detectCluster() )

performance <- microbenchmark (

sma\_11 <- sma\_parallel(period, symbol, data, cluster).

unit=”us”

)

all(sma\_1$sma-sma\_11 <= 0.001, na.rm = TRUE)

#> TRUE

summary(performance)$median

#> [1] 44825.9355

W tym przypadku nasza mediana czasu to 44,825. 9355 mikrosekundy, czyli mniej więcej gorzej niż byliśmy w stanie osiągnąć ze sma\_slow\_2(). Czy zrównoleglenie nie miało być znacznie szybsze? Odpowiedź brzmi: tak, podczas pracy z większymi nakładami. Kiedy używamy danych, które mają miliony obserwacji (a nie 100 obserwacji, których używaliśmy do tych testów), będzie to szybsze, ponieważ czas ich wykonania nie wydłuży się tak bardzo, jak w przypadku innych wdrożeń. W tej chwili sma\_parallel() płaci duży stały koszt, który nie jest dobrą inwestycją w przypadku pracy z małymi zbiorami danych, ale gdy zaczynamy pracować z większymi zbiorami danych, stały koszt zaczyna być mały w porównaniu ze wzrostem wydajności. Aby sfinalizować sekcję, pamiętaj, aby wywołać stopCluster(cluster), gdy chcesz przestać korzystać z klastra. W takim przypadku zostawimy ją uruchomioną, ponieważ będziemy nadal wykonywać więcej testów porównawczych.

str. 43

Używanie C++ i Fortran do przyspieszenia obliczeń

Czasami kod R nie jest wystarczająco szybki. Czasami używałeś profilowania, aby dowiedzieć się, gdzie są twoje wąskie gardła i zrobiłeś wszystko, o czym możesz pomyśleć w R, ale twój kod nadal nie jest wystarczająco szybki. W takich przypadkach użyteczną alternatywą może być delegowanie niektórych części implementacji do bardziej wydajnych języków, takich jak Fortran i C++. Jest to zaawansowana technika, która często może okazać się bardzo przydatna, jeśli wiesz, jak programować w takich językach. Delegowanie kodu do innych języków może rozwiązać wąskie gardła, takie jak:

\* Pętle, których nie można łatwo wektoryzować ze względu na zależności iteracyjne

\* Procesy, które obejmują wywoływanie funkcji miliony razy

\* Nieefektywne, ale niezbędne struktury danych, które są powolne w R

Delegowanie kodu do innych języków może zapewnić duże korzyści w zakresie wydajności, ale wiąże się również z kosztem bycia bardziej przejrzystym i ostrożnym w przypadku typów obiektów, które są przenoszone. W R możesz uciec od prostych rzeczy, takich jak nieprecyzyjne określenie liczby będącej liczbą całkowitą lub rzeczywistą. W tych innych językach nie możesz; każdy obiekt musi mieć określony typ i pozostaje niezmienny przez całą realizację.

str. 44

Korzystanie ze starej szkoły w Fortran

Zaczniemy od starej szkoły, używając najpierw języka Fortran. Jeśli go nie znasz, Fortran jest najstarszym nadal używanym językiem programowania. Został zaprojektowany do wykonywania wielu obliczeń bardzo wydajnie i przy niewielkich zasobach. Opracowano wiele bibliotek numerycznych i wiele wysokowydajnych systemów obecnie nadal go używają, bezpośrednio lub pośrednio. Oto nasza implementacja, nazwana sma\_foortran(). Składnia może Cię zaskoczyć, jeśli nie jesteś przyzwyczajony do pracy z kodem Fortran, ale jest wystarczająco prosty do zrozumienia. Po pierwsze, zwróć uwagę, że aby zdefiniować funkcję technicznie znaną jako subroutine w języku Fortran, używamy słowa kluczowego subroutine przed nazwą funkcji. Podobnie jak nasze poprzednie implementacje, otrzymuje period i data (używamy nazwy dataa z dodatkiem a na końcu, ponieważ Fortran ma zarezerwowane słowo kluczowe data, którego nie powinniśmy używać w tym przypadku) i założymy, że dane są już filtrowane pod kątem prawidłowych symboli w tym miejscu. Następnie zwróć uwagę, że wysyłamy nowe argumenty, których wcześniej nie wysyłaliśmy, a mianowicie smas i n. Fortran jest specyficznym językiem w tym sensie, że nie zwraca wartości, zamiast tego używa efektów ubocznych. Oznacza to, że zamiast oczekiwać czegoś z powrotem po wywołaniu podprogramu w języku Fortran, powinniśmy oczekiwać, że ten podprogram zmieni jeden z przekazanych do niego obiektów i powinniśmy traktować to jako naszą wartość return. W tym przypadku smas spełnia tę rolę; początkowo zostanie wysłany jako tablica niezdefiniowanych wartości rzeczywistych, a celem jest zmodyfikowanie jego zawartości za pomocą odpowiednich wartości SMA. Wreszcie n, reprezentuje liczbę elementów w przesyłanych przez nas danych. Klasyczny Fortran nie ma sposobu na określenie rozmiaru przekazywanej do niego tablicy i wymaga od nas ręcznego określenia rozmiaru; dlatego musimy wysłać n. W rzeczywistości istnieją sposoby obejścia tego problemu, ale ponieważ nie jest to tekst o języku Fortran, postaramy się, aby kod był tak prosty, jak to tylko możliwe. Następnie zwróć uwagę, że musimy zadeklarować typ obiektów, z którymi mamy do czynienia, a także ich rozmiar w przypadku, gdy są to tablice. Przechodzimy do deklaracji pos (która zajmuje miejsce pozycji w naszej poprzedniej implementacji, ponieważ Fortran nakłada ograniczenie na długość każdej linii, której nie chcemy naruszać), n, endd (ponownie end jest to słowo kluczowe w Fortranie, więc użyj zamiast tego nazwy endd) i period jako liczby całkowite. Deklarujemy również dataa(n), smas(n) i sma jako liczby rzeczywiste, ponieważ będą one zawierać części dziesiętne. Zauważ, że określamy rozmiar tablicy z częścią (n) w pierwszych dwóch obiektach. Po zadeklarowaniu wszystkiego, czego będziemy używać, kontynuujemy naszą logikę. Najpierw tworzymy pętlę for , która jest wykonywana za pomocą słowa kluczowego do w języku Fortran, po którym następuje unikalny identyfikator (który zwykle jest nazywany wielokrotnością dziesiątek lub setek), nazwa zmiennej, która będzie używana do iteracji oraz wartości, które przyjmie endd i 1 do n , w tym przypadku odpowiednio. Wewnątrz pętli for przypisujemy pos, które być równe endd i sma równe 0, tak jak to zrobiliśmy w niektórych naszych poprzednich implementacjach. Następnie tworzymy pętlę while z kombinacją słów kluczowych do … while i podajemy warunek, który należy sprawdzić, aby zdecydować, kiedy z niej wyjść. Zauważ, że Fortran używa zupełnie innej składni dla operatorów porównania. W szczególności operator .lt. oznacza mniejszy niż, podczas gdy operator .ge. oznacza większe niż lub równe. Jeśli którykolwiek z dwóch określonych warunków nie zostanie spełniony, zakończymy pętlę while . To powiedziawszy, reszta kodu powinna być oczywista. Jedyną inną niezwykłą właściwością składni jest wcięcie kodu do szóstej pozycji. To wcięcie ma znaczenie w języku Fortran i powinno pozostać takie, jakie jest. Ponadto identyfikatory liczb podane w pierwszych kolumnach kodu powinny być zgodne z odpowiednimi mechanizmami zapętlenia i powinny znajdować się po lewej stronie kodu logicznego. Aby uzyskać dobre wprowadzenie do języka Fortran, możesz zapoznać się z samouczkiem Stanforda dotyczącym języka Fortran 77 (). Powinieneś wiedzieć, że istnieją różne wersje Fortrana, a wersja 77 jest jedną z najstarszych. Jednak jest to również jeden z lepiej obsługiwanych:

subroutine sma\_fortran(period, dataa, smas, n)

integer pos, n , endd, period

real dataa(n), smas(n)., sma

do 10 endd = 1 , n

pos = endd

sma = 0.0

do 20 while ((endd – pos .t. period) .and. (pos .ge. 1))

sma = sma + dataa(pos)

pos = pos – 1

end do

if (endd – pos .eq. period) then

sma = sma / period

else

sma = 0

 end if

smas(endd) = sma

10 continue

end

Gdy kod jest gotowy, musisz go skompilować, zanim będzie można go wykonać w R. Kompilacja to proces tłumaczenia kodu na instrukcje na poziomie maszyny. Masz dwie opcje podczas kompilowania kodu w języku Fortran: możesz to zrobić ręcznie poza R lub możesz to zrobić w R. Druga opcja jest zalecana, ponieważ możesz skorzystać z narzędzi R. Jednak pokazujemy oba z nich. Pierwszą można osiągnąć za pomocą następującego kodu:

$ gfortran -c sma-delegated-fortran.f -o sma-delegated-fortran.so

Ten kod powinien zostać wykonany w terminalu Bash (który można znaleźć w systemach operacyjnych Linux lub Mac). Musimy upewnić się, że mamy zainstalowany kompilator, który prawdopodobnie został zainstalowany, gdy R był. Następnie wywołujemy go, mówiąc mu, aby skompilował (używając opcji -c) plik sma-delegated-fortran.f (który zawiera kod Fortran, który pokazaliśmy wcześniej) i udostępnił plik wyjściowy (z opcją -o) o nazwie sma-delegated-fortran.so. Naszym celem jest zdobycie tego pliku .so, czyli tego, czego potrzebujemy w R do wykonania kodu Fortran. Sposobem kompilacji w R, który jest zalecany, jest użycie następującego wiersza:

system(„R MD SHLIB sma-delegated-fortran.f”)

Zasadniczo mówi R, aby wykonał polecenie, które tworzy bibliotekę współdzieloną pochodzącą z pliku sma-delegated-fortran.f. Zwróć uwagę, że funkcja system() po prostu wysyła otrzymany ciąg do terminala w systemie operacyjnym, co oznacza, że ​​mogłeś użyć tego samego polecenia w terminalu Bash, który został użyty do ręcznej kompilacji kodu. Aby załadować udostępnioną bibliotekę do pamięci R, używamy funkcji dyn.load(), podając lokalizację pliku .so, którego chcemy użyć, i faktycznie wywołujemy udostępnioną bibliotekę, która zawiera implementację , używamy funkcji Fortran(). Ta funkcja wymaga jawnego wykonania sprawdzenia typu i wymuszenia przez użytkownika przed jej wywołaniem. Aby zapewnić podobny podpis, jak ten dostarczany przez poprzednie funkcje, utworzymy funkcję o nazwie sma-delegated-fortran(), która odbiera parametry period ,symbol i data i tak jak poprzednio, a także filtruje dane tak jak wcześniej, oblicza długość danych i wstawia w n w programie i używa tej funkcji .Fortran() do wywołania podprogramu sma\_fortran(), dostarczając odpowiednie parametry. Zwróć uwagę, że otaczamy parametry wokół funkcji, które wymuszają typy tych obiektów zgodnie z wymaganiami naszego kodu w języku Fortran. Lista result utworzona przez funkcję .Fortran() zawiera obiekty period, dataa, smas i n odpowiadające parametrom wysłanym do podprogramu, z zawartością pozostawioną w nich po wykonaniu podprogramu. Jak wspomnieliśmy wcześniej, interesuje nas zawartość obiektu sma, ponieważ zawierają one wartości, których szukamy. Dlatego wysyłamy tylko tę część z powrotem po przekonwertowaniu jej na typ numeric w R. Transformacje, które widzisz przed wysłaniem obiektów do Fortran i po ich odzyskaniu, są czymś, z czym musisz być bardzo ostrożny. Na przykład, jeśli zamiast używać single() i as.single(data) używamy double(n) i as.double(data), nasza Implementacja Fortran nie zadziała. To jest coś, co można zignorować w R, ale nie można tego zignorować w przypadku Fortrana:

system(„R MD SHLIB sma-delegated-fortran.f”)

dyn.load(„sma-delegated-fortran.so”)

sma\_delegated\_fortran <- function(period ,symbol, data) {

data <- data[which(data$symbol == symbol), „pric\_usd”]

n <- length(data)

results <- .Fortan(

„sma\_fortran”,

period = as.integer(period),

dataa = as.single (data),

smas = single(n),

n – as.integer(n)

)

return(as.numeri(result$smas))

}

Tak jak wcześniej, porównujemy i testujemy poprawność:

performance <- microbenchmark (

sma\_12 <- sma\_delegated\_fortran(period, symboo, data),

unit = „us”

)

all(sma\_1$sma – sma\_12 <= 0.001, na.rm = TRUE)

#> TRUE

summary(perofmane)$median

#> [1] 148.0335

W tym przypadku nasz średni czas wynosi 148.0335 mikrosekundy, co sprawia, że ​​jest to najszybsza implementacja do tego momentu. Zwróć uwagę, że minęło niewiele ponad połowa czasu od najbardziej wydajnej implementacji, jaką mogliśmy wymyślić, używając tylko R. Spójrz na poniższą tabelę:



str. 48

Korzystanie z nowoczesnego podejścia w C++

Teraz pokażemy, jak zastosować bardziej nowoczesne podejście przy użyciu C++. Celem tej sekcji jest dostarczenie informacji wystarczających do rozpoczęcia samodzielnego eksperymentowania z C++ w języku R. Przyjrzymy się tylko niewielkiemu fragmentowi tego, co można zrobić, łącząc R z C++ za pośrednictwem pakietu Rcpp (który jest domyślnie instalowany w R), ale powinno wystarczyć, aby zacząć. Jeśli nigdy nie słyszałeś o C++, jest to język używany głównie w przypadku ograniczeń zasobów ważna rola i optymalizacja wydajności ma ogromne znaczenie. Zanim przejdziemy dalej, upewnij się, że masz w swoim systemie kompilator C++. W systemie Linux powinieneś móc używać gcc. Na Macu powinieneś zainstalować Xcode ze sklepu z aplikacjami. W systemie Windows należy zainstalować Rtools. Po przetestowaniu kompilatora i upewnieniu się, że działa, powinieneś być w stanie postępować zgodnie z tą sekcją. C++ jest bardziej czytelny niż kod Fortran, ponieważ jest zgodny z większą liczbą konwencji składniowych, do których jesteśmy przyzwyczajeni. Jednak tylko dlatego, że przykład, którego użyjemy, jest czytelny, nie myśl, że C++ jest ogólnie łatwym w użyciu językiem; to nie jest. Jest to język na bardzo niskim poziomie, a jego prawidłowe używanie wymaga dużej ilości wiedzy. Powiedziawszy to, zacznijmy. Linia #include służy do przenoszenia definicji zmiennych i funkcji z języka R do tego pliku podczas kompilacji. Dosłownie zawartość pliku Rcpp.h jest wklejana dokładnie tam, gdzie znajduje się instrukcja inlude. Pliki kończące się rozszerzeniami .h nazywane są plikami nagłówkowymi i służą do dostarczania niektórych typowych definicji między użytkownikiem kodu a jego twórcami. Odgrywają podobną rolę do tego, co nazwaliśmy interfejsem poprzednio. Linia using namespace Rcpp pozwala na użycie krótszych nazw funkcji. Zamiast określać Rcpp : : NumericVector, możemy po prostu użyć NumericVector do zdefiniowania typu obiektu data. Zrobienie tego w tym przykładzie może nie być zbyt korzystne, ale kiedy zaczniesz programować dla złożonego kodu C++, naprawdę się to przyda. Następnie zauważysz kod //[[Rcpp::export(sma\_delegated\_cpp)]]. Jest to znacznik oznaczający funkcję tuż pod nim, aby R wiedział, że powinien ją zaimportować i udostępnić w kodzie R. Argument wysłany do export() to nazwa funkcji, która będzie dostępna w R i niekoniecznie musi odpowiadać nazwie funkcji w C++. W tym przypadku sma\_delegated\_cpp()będzie to funkcja, którą wywołujemy w R i wywoła funkcję smaDelegated() w C++:

#include

using namespae Rcpp();

//[[Rcpp::export(sma\_delegated\_cpp)]]

NumericVecto smaDelegated(int period, NumericVector data) {

int position , n = data.size();

NumericVector result(n);

double sma;

for (int end = 0; end < n; end++) {

position = end;

sma = 0;

while (end – position < period && position >= 0) {

sma = sma + data[position];

positin = position -1;

}

if (end – position ==period) {

sma = sma / period;

} else {

sma = NA\_REAL;

}

result[end] = sma;

}

return result;

}

Następnie wyjaśnimy rzeczywistą funkcję smaDelegates(). Ponieważ masz dobre pojęcie o tym, co robi w tym momencie, nie wyjaśnimy jego logiki, tylko składnię, która nie jest tak oczywista. Pierwszą rzeczą, na którą należy zwrócić uwagę, jest to, że przed nazwą funkcji znajduje się słowo kluczowe, które jest typem wartości return funkcji. W tym przypadku jest to NumericVetor, co jest zawarte w pliku Rcpp.h. Jest to obiekt przeznaczony do łączenia wektorów między językami R i C++. Inne typy wektorów udostępniane przez Rcpp.h to,IntegerVector, LogicalVector i CharaterVector. Masz również IntegerMatrix, NumericMatrix, LogicalMatrix i CharacterMatrix dostępne. Następnie należy zauważyć, że parametry otrzymane przez funkcję mają również skojarzone z nimi typy. W szczególności period jest liczbą całkowitą (int)a data jest NumericVector, tak jak wynik funkcji. W tym przypadku nie musieliśmy przekazywać obiektów output lub length, tak jak to zrobiliśmy z Fortranem. Ponieważ funkcje w C++ mają wartości wyjściowe, ma również dość łatwy sposób obliczania długości obiektów. Pierwsza linia funkcji deklaruje zmienne position i n, i przypisuje długość danych do drugiej. Możesz używać przecinków, tak jak my, do deklarowania różnych obiektów tego samego typu jeden po drugim zamiast dzielenia deklaracji i przypisań na osobne wiersze. Deklarujemy również wektor result z długością n; zwróć uwagę, że notacja ta jest podobna do notacji Fortran. Wreszcie, zamiast używać słowa kluczowego real, tak jak robimy to w Fortranie, używamy float lub double w celu oznaczenia takich liczb. Z technicznego punktu widzenia istnieje różnica w precyzji dozwolonej przez takie słowa kluczowe i nie są one zamienne, ale nie będziemy się tym martwić. Reszta funkcji powinna być jasna, z wyjątkiem może przypisania sma = NA\_REAL. Ten obiekt NA\_REAL jest również udostępniany przez Rpp jako sposób na oznaczenie, co powinno zostać wysłane do R jako NA. Wszystko inne powinno wyglądać znajomo. Teraz, gdy nasza funkcja jest gotowa, zapisujemy ją w pliku o nazwie sma-delegated-cpp.cpp i używamy funkcji R sourceCpp, aby skompilować ją za nas i przenieść do R. Rozszerzenie .cpp oznacza zawartość napisaną w języku C++. Należy pamiętać, że funkcji przeniesionych do języka R z plików C++ nie można zapisać w pliku .Rdata do późniejszej sesji. C++ ma być bardzo zależny od sprzętu, na którym jest skompilowany, a zrobienie tego prawdopodobnie spowoduje różne błędy. Za każdym razem, gdy chcesz użyć funkcji C++, powinieneś ją skompilować i załadować funkcją sourceCpp()w momencie użycia.

library(Rcpp)

sourceCpp(„./sma-delegated-cpp.cpp”)

sma\_delegated\_cpp <- function(period, symbol, datat) {

data <- as.numeric(data[which(data$symbol == symbol), „price\_usd”])

return(sma\_pp(period, data))

}

Jeśli wszystko działało poprawnie, nasza funkcja powinna być użyteczna w R, więc wykonujemy testy porównawcze i testy poprawności. Obiecuję, że to ostatnia:

performance <- microbenchamrk (

sma\_13 <- sma\_delegated\_cpp(period, symboo, data),

unit = „us”

)

all(sma\_1$sma = sma\_13 <= 0.001, na.rm = TRUE)

#> TRUE

summary(performance)$median

#>[1] 80.6415

Tym razem nasz średni czas wyniósł 80.6415 mikrosekundy, czyli o trzy rzędy wielkości szybciej niż w naszej pierwszej implementacji. Pomyśl o tym w ten sposób: jeśli podasz dane wejściowe dla sma\_delegated\_cpp(), których wykonanie zajęło około godziny, sma\_slow\_1() zajmie to około 1000 godzin, czyli około 41 dni. Czy to nie zaskakująca różnica? Kiedy jesteś w sytuacjach, które zajmują tyle czasu na wykonanie, zdecydowanie warto spróbować i zoptymalizować swoje implementacje. Możesz użyć tej funkcji cppFUnction() do napisania kodu C++ bezpośrednio w pliku .R, ale nie powinieneś tego robić. Zachowaj to tylko do testowania małych fragmentów kodu. Rozdzielenie implementacji C++ na osobne pliki pozwala wykorzystać moc wybranego edytora (lub IDE) do prowadzenia użytkownika przez proces programowania, a także do dokładniejszego sprawdzania składni.

str. 52

Patrząc wstecz na to, co osiągnęliśmy

Jak wiecie, do tej pory testowaliśmy nasz kod przy użyciu podzbioru danych, który zawiera tylko pierwsze 100 obserwacji. Jednak, jak widzieliśmy na początku , wydajność może się różnić dla różnych implementacji, w zależności od rozmiaru danych wejściowych. Aby zebrać wszystkie nasze wysiłki w tym rozdziale, utworzymy kilka funkcji, które pomogą nam zmierzyć, jak zmieniają się czasy wykonania naszych implementacji, gdy korzystamy z większej liczby obserwacji z naszych danych. Najpierw wprowadzamy nasze wymagania do R, głównie pakiety microbenchmark i ggplot2 oraz pliki, które zawierają nasze implementacje. Następnie tworzymy funkcję sma\_performance(), która przyjmuje symbol, period, original\_data, listę nazwaną sizes, której elementy są liczbą obserwacji, które zostaną pobrane z original\_data do przetestowania naszych implementacji, cluster aby uniknąć narzutu inicjowania jej w naszej funkcji sma\_parallel(), jak widzieliśmy w odpowiedniej sekcji i ile razy chcemy mierzyć każdą implementację. Jak widać, dla każdego rozmiaru w rozmiarach bierzemy odpowiednią liczbę obserwacji w obiekcie data i wysyłamy ją wraz z innymi niezbędnymi argumentami dla funkcji sma\_microbenchmark(). Następnie dodajemy wartość size do ramki danych result, która jest zapewniana przez funkcję summary() zastosowaną na wierzchu obiektu wynikowego z funkcji mirobenchmar() z sma\_microbenchmark(). Musimy to dodać sami, ponieważ funkcja nie ma żadnej wiedzy na temat rozmiaru danych, z którymi ma do czynienia. Na koniec spłaszczamy listę ramek danych na liście results za pomocą funkcji do.call(„rbind”, results), która wysyła pojedynczą ramkę danych jako dane wyjściowe. Funkcja sma\_icrobenchmark() jest bardzo prosta. Otrzymuje tylko niektóre parametry i przekazuje je dalej do każdej implementacji, która będzie mierzona przez funkcję. Zwróć uwagę, że zostawiamy wewnątrz funkcję sma\_paralel)inefficient(), ale jest ona zakomentowana, aby uniknąć problemów ze skalą na wykresie, który w końcu utworzymy (ponieważ jest bardzo powolny, wypaczy nasz wykres). Wynikowy obiekt funkcji sma\_performance() zwraca ramkę danych z wynikami dla wszystkich testów, która jest używana jako dane wejściowe dla funkcji graph\_sma\_performance() w postaci obiektów results. Otrzymuje również sizes, który zostanie użyty do zdefiniowania wartości na osi x. Jak widać, wywołujemy remove\_arguments(), o czym wspominamy, idąc naprzód. Tworzy wykres przy użyciu funkcji ggplot(), geom\_point() i geom\_line(), jak widzieliśmy wcześniej, i używamy skal logarytmicznych dla obu osi. Funkcja remove\_arguments() robi dokładnie to, co mówi, usuwa nawiasy i argumenty z wywołań funkcji, dzięki czemu zachowujemy tylko nazwę funkcji. Ma to na celu zmniejszenie miejsca w legendzie wykresu. Aby to osiągnąć, używamy funkcji gsub(), którą widzieliśmy wcześniej. Aby użyć przedstawionego kodu, po prostu tworzymy listę sizes, której brakuje i używamy wszystkich innych obiektów, które zdefiniowaliśmy wcześniej. W tym konkretnym przypadku chcemy zmierzyć pierwsze 10, 100, 1000 i 10 000 obserwacji. Jeśli chcesz, możesz zwiększyć tę listę o większe kwoty. Pamiętaj, że całkowita liczba obserwacji w symulowanych danych to nieco ponad 1 000 000. Powstały wykres przedstawia liczbę obserwacji na osi x i medianę w mikrosekundach na osi y. Obie osie używają skali logarytmicznej, więc miej na uwadze interpretację relacji. Jak widać, gdy rozmiar danych wejściowych jest mniejszy (po lewej stronie wykresu), różnica czasu wykonania jest mniejsza, a gdy zwiększamy rozmiar wejściowy, różnice zaczynają być coraz większe, szczególnie biorąc pod uwagę skale logarytmiczne. Oto kilka interesujących rzeczy, na które należy zwrócić uwagę:

* Funkcja sma\_efficient\_1 okazała się wolniejsza niż sma\_slow\_7() w przypadku 100 obserwacji, w rzeczywistości jest szybsza przy użyciu 10 000 obserwacji. To pokazuje, że kompromis miał sens, zwłaszcza w miarę wzrostu nakładów.
* sma\_efficient\_2() : Ta implementacja jest szybsza, dla 10 obserwacji, niż implementacja Fortran. To dość zaskakujące i pokazuje, że narzut związany z wywołaniem kodu w języku Fortran nie jest tego wart przy takim rozmiarze danych wejściowych. Jednak sma\_efficient\_2() szybko staje się wolniejszy wraz ze wzrostem rozmiaru wejściowego.
* sma\_paralle(): Ta implementacja jest powolna z powodu całego narzutu, jaki ponosi, jak widzieliśmy w odpowiedniej sekcji, ale jest to również implementacja, w której procentowy wzrost czasu jest najmniejszy, gdy zwiększa się rozmiar danych wejściowych. To powinno nas skłonić do zastanowienia się, co się dzieje, gdy mamy do czynienia z pełnymi danymi? Czy w tym momencie będzie szybciej niż implementacje Fortran lub C ++, które wydają się rosnąć szybciej? Pozostawiamy to jako ćwiczenie dla czytelnika.

Na koniec, dla ciekawskiego czytelnika, jak myślisz, co się stanie, jeśli użyjesz implementacji sma\_delegated\_cpp() wraz z podejściem zrównoleglania jakie pokazaliśmy? Jeśli chcesz poznać odpowiedź, zdecydowanie powinieneś spróbować samemu.

str. 54

Inne tematy związane z poprawą wydajności

Zobaczyliśmy przegląd najważniejszych i najczęściej używanych technik optymalizacji implementacji języka R. Jednak wciąż jest wiele rzeczy, których nie omówiliśmy. W kolejnych sekcjach pokrótce omówimy niektóre z nich.

Wstępne przydzielanie pamięci w celu uniknięcia powielania

Wstępna alokacja pamięci jest ważną techniką, którą omawialiśmy niejawnie, kiedy używaliśmy funkcji lapply(), ponieważ wykonuje ona za nas prealokację. Przydatne może być jednak bardziej wyraźne wyjaśnienie. Jak już widzieliśmy, dynamicznie rosnące obiekty w R nie są świetne pod względem wydajności. Zamiast tego należy zdefiniować obiekt o pełnym rozmiarze, jakiego będziesz potrzebować, a następnie wykonać aktualizacje jego elementów zamiast ich ponownego tworzenia. Aby to osiągnąć, możesz użyć czegoś podobnego do double(10) dla zdefiniowania wektora double, który będzie zawierał najwyżej 10 elementów. Ilekroć określisz rozmiar obiektu przed rozpoczęciem korzystania z niego, pomoże ci to uniknąć ponownego tworzenia nowych obiektów za każdym razem, gdy jego rozmiar zostanie zwiększony i zaoszczędzi ci dużo czasu. Jednak dokładna alokacja wstępna nie zawsze jest możliwa, ponieważ wymaga znajomości całkowitej liczby przed iteracją. Czasami możemy tylko poprosić o wielokrotne przechowywanie wyniku, nie znając dokładnej całkowitej liczby. W takim przypadku może nadal dobrym pomysłem jest wstępne przydzielenie listy lub wektora o rozsądnej długości. Po zakończeniu iteracji, jeśli liczba iteracji nie osiągnie wstępnie przydzielonej długości, możemy wziąć podzbiór listy lub wektora. W ten sposób możemy uniknąć intensywnej realokacji struktur danych. Jeśli chodzi o wstępne przydzielanie pamięci, R nie różni się od innych języków programowania. Jednak będąc językiem interpretowanym, nakłada mniej ograniczeń; w ten sposób użytkownicy mogą łatwo przeoczyć tego typu problemy. R nie zgłosi żadnego błędu kompilacji, jeśli pamięć wektora nie jest wstępnie przydzielona. Należy o tym pamiętać podczas pisania szybkiego kodu.

str. 55

Trochę szybsze tworzenie kodu R dzięki kompilacji kodu bajtowego

Mimo że R jest językiem interpretowanym, może przejść przez krótką fazę przed wykonaniem kodu zwaną kompilacją kodu bajtowego, która jest mniej rygorystyczną procedurą kompilacji. W niektórych scenariuszach może zaoszczędzić od 5% do 10% czasu, jeśli już zoptymalizowane funkcje nie są intensywnie używane. Wszystkie podstawowe funkcje języka R są domyślnie kompilowanymi kodami bajtowymi. Aby kompilować funkcje w kodzie bajtowym, po załadowaniu pakietu compiler ,należy użyć funkcji cmpfunc() owiniętej wokół funkcji, którą chcesz skompilować. Możesz również przesłać argumenty optios, takie jak options = list(optimize=3)), gdzie element optymalizacji powinien być liczbą całkowitą z przedziału od 0 do 3. Im wyższa liczba, tym więcej wysiłku R włoży w optymalizację kompilacji. Poniższe wiersze pokazują, jak utworzyć funkcję sma\_efficient\_2\_compiled(), która jest skompilowaną wersją funkcji sma\_efficient)2():

library(compiler)

sma)efficient\_2\_compiled <-

cmpfun(sma\_efficient\_2, options = list(optimize = e))

str. 55

Kompilacja Just-in-time (JIT) kodu R.

R obsługuje również kompilację Just-in-Time (JIT). Gdy kompilacja JIT jest włączona, język R automatycznie kompiluje bajtami kod, który jest wykonywany bez jawnego wywoływania jednej z funkcji kompilacji. Aby aktywować kompilację JIT, użyj funkcji enableJIT(). Argument level mówi R, ile kodu należy skompilować przed wykonaniem; 0 wyłącza JIT, 1 kompiluje funkcje przed ich pierwszym użyciem, 2 kompiluje także funkcje przed ich zduplikowaniem, a 3 także kompiluje pętle przed ich wykonaniem:

library(compiler)

enableJIT(level=3)

Kompilację JIT można również włączyć, ustawiając środowisko R\_ENABLE\_JIT w systemie operacyjnym przed uruchomieniem R. Wartość R\_ENABLE\_JIT należy ustawić na wartość argumentu level.

str 56

Korzystanie z zapamiętywania lub warstw pamięci podręcznej

Jeśli masz algorytmy deterministyczne, za każdym razem, gdy zapewniasz równe dane wejściowe, powinieneś otrzymywać równe wyniki, a jeśli tak jest, a proces przejścia od danych wejściowych do wyjściowych jest bardzo czasochłonny, możesz użyć zapamiętywania lub warstw pamięci podręcznej. Podstawową ideą jest to, że przechowujesz kilka kopii danych wejściowych i wyjściowych, a za każdym razem, gdy dane wejściowe są wysyłane do funkcji, przed obliczeniem danych wyjściowych, sprawdzasz, czy dane wyjściowe tego konkretnego wejścia zostały wcześniej obliczone. Jeśli tak, wyślij to, zamiast wykonywać całą pracę ponownie. Oznacza to, że dane wyjściowe dla każdego wejścia należy obliczać tylko raz. Powinieneś spróbować zaimplementować taką warstwę w funkcji fibonacci\_recursive() utworzoną na początku , aby zobaczyć, jak duży wpływ mogą mieć tego rodzaju techniki, nawet przy użyciu powolnych algorytmów. Czasami tego typu techniki są również używane, nawet jeśli dane wyjściowe dla danego wkładu zmieniają się w czasie. Wszystko, co musisz zrobić w takich przypadkach, to zapewnić mechanizm, który unieważni lub usunie przechowywaną relację wejście / wyjście po określonym czasie, tak aby została ponownie obliczona następnym razem, gdy dane wejściowe zostaną użyte.

str. 56

Poprawa zarządzania danymi i pamięcią

R, jak każdy język programowania, jest ograniczony przez procesor, pamięć RAM i wejścia / wyjścia, a my skupiliśmy się na zwiększeniu szybkości części procesora. Jednak znaczny wzrost wydajności można osiągnąć, zwiększając efektywność wykorzystania pamięci RAM i we / wy. Pomiar wykorzystania pamięci RAM (pamięci) najlepiej wykonywać poza językiem R przy użyciu narzędzi dostarczonych przez system operacyjny do tego właśnie celu. Informacje, które raportują te narzędzia, różnią się w zależności od systemu operacyjnego, ale oto kluczowe wskaźniki, na które należy zwrócić uwagę: użycie procesora, wolna pamięć, pamięć fizyczna, rozmiar wymiany i bajty odczytu / zapisu na sekundę. Jeśli napotkasz wysokie wykorzystanie procesora, procesor jest prawdopodobnie głównym wąskim gardłem dla wydajności R. Użyj technik profilowania opisanych w tym rozdziale, aby zidentyfikować, które części kodu zajmują większość czasu procesora. Jeśli napotkasz wystarczającą ilość wolnej pamięci systemowej z wysokim poziomem operacji we / wy dysku, kod prawdopodobnie wykonuje wiele operacji odczytu / zapisu na dysku. Usuń wszystkie niepotrzebne operacje we / wy i przechowuj dane pośrednie w pamięci, jeśli jest wystarczająca ilość pamięci. Jeśli napotkasz niskie wykorzystanie procesora i małą ilość wolnej pamięci systemowej przy dużym rozmiarze wymiany, prawdopodobnie systemowi zabraknie pamięci fizycznej i tym samym wymienia pamięć na dysk. W takim przypadku sprawdź, czy masz wystarczająco dużo zasobów, aby obsłużyć obciążenia, które wysyłasz do R, a jeśli to zrobisz, spróbuj użyć funkcji rm(), aby usunąć nieużywane obiekty, które czekają na pamięć z sesji R. Jeśli napotkasz scenariusz podobny do poprzedniego, ale wiesz, że nie masz wystarczającej ilości pamięci, aby obsłużyć pełne dane, z którymi pracujesz, nawet jeśli zrobiłeś to skutecznie, możesz spróbować podzielić swoje dane na partycje. Czy możesz pracować z podzbiorem danych według części, a następnie połączyć wyniki? Jeśli tak, powinieneś spróbować. Na przykład, jeśli pełne dane nie mieszczą się w pamięci i próbujesz znaleźć maksymalną wartość, możesz podzielić dane na cztery części, załadować je po kolei, obliczyć maksimum dla każdej z nich, i usuń je z pamięci po wykonaniu tej czynności, zachowując maksimum, a następnie uzyskując maksimum z czterech maksimów obliczonych oddzielnie. Inną możliwością dla scenariusza takiego jak poprzedni jest po prostu migracja obsługi danych do bazy danych. Bazy danych są wyspecjalizowanymi narzędziami do przetwarzania danych i pozwalają uniknąć wąskiego gardła w R, ponieważ tylko wstępnie przetworzony podzbiór potrzebnych danych jest wprowadzany do R. Większość baz danych obecnie wykonuje również bardzo wydajne proste operacje, takie jak znajdowanie maksimum.

str. 57

Korzystanie ze specjalistycznych pakietów do wydajności

Innym dobrym sposobem na zwiększenie wydajności implementacji jest wyszukanie wyspecjalizowanych funkcji opublikowanych w pakietach CRAN lub gdzie indziej. Zanim przejdziesz do zmiany własnego kodu, spójrz i zobacz, czy możesz znaleźć bardzo wydajną implementację w innym miejscu. Istnieją ogromne różnice w jakości i szybkości pakietów CRAN, ale ich wykorzystanie może zdecydowanie zaoszczędzić sporo czasu. Dwa bardzo wydajne pakiety, które pomogą Ci opracować wydajne implementacje, to pakiety dta.table i dplyr. Mogą zapewnić wydajne sposoby radzenia sobie z ramkami danych, a także w przypadku dplyr innych obiektów. Biblioteka podprogramów podstawowej algebry liniowej (BLAS) może być również bardzo pomocna podczas wykonywania operacji algebry liniowej. Jest napisany przy użyciu języka Fortran i jest wysoce zoptymalizowany.

str. 58

Elastyczność i moc dzięki przetwarzaniu w chmurze

Czasami nie potrzebujesz nawet większej mocy obliczeniowej ani efektywnego wykorzystania zasobów. Czasami wystarczy uruchomić R na innym komputerze bez wiązania własnego przez wiele godzin lub dni. W takich przypadkach korzystanie z zasobów przetwarzania w chmurze może być bardzo przydatne. Zasoby przetwarzania w chmurze są przydatne nie tylko wtedy, gdy chcesz użyć dodatkowej maszyny, ale są bardzo wydajnym sposobem na pozyskanie superkomputerów do wykonania pracy za Ciebie. Bardzo łatwo jest zbudować maszynę z 64 rdzeniami procesora i 512 GB pamięci RAM. Korzystanie z takiego systemu może być tańsze niż myślisz i może być wykorzystane do bardzo kosztownych obliczeń, które zajęłyby zbyt dużo czasu w standardowym sprzęcie.

str. 58

Wyspecjalizowane dystrybucje R.

Wreszcie, jeśli żadna z poprzednich opcji nie zadziałała, możesz również użyć wyspecjalizowanych dystrybucji R. Te dystrybucje są utrzymywane niezależnie od zwykłej dystrybucji R i koncentrują się na optymalizacji określonych aspektów w R. Niektóre z nich są zbudowane w celu zapewnienia precyzyjnej kontroli równoległości, a inne robią to automatycznie. Nauka korzystania z tych dystrybucji może wymagać dużo czasu, co może być korzystne lub nie w Twoim konkretnym przypadku.

str. 58

Podsumowanie

Zobaczyliśmy najważniejsze przyczyny powolnego kodu R: programowanie bez zrozumienia niezmienności obiektów, charakter interpretowanych typów dynamicznych, procesy związane z pamięcią i procesy jednowątkowe. Dowiedzieliśmy się, że pierwszy można zredukować za pomocą odpowiedniego R, drugi można zredukować, delegując do języków statystycznie typowanych, takich jak Fortran lub C ++, trzeci można zredukować za pomocą mocniejszych komputerów (szczególnie z większą ilością pamięci RAM), oraz wreszcie czwarty można zredukować za pomocą równoległości. Wspomnieliśmy również o niektórych zmiennych, które możemy chcieć wziąć pod uwagę przy podejmowaniu decyzji, czy optymalizować nasze implementacje, jak mała różnica we wdrożeniu może skutkować dużymi ulepszeniami wydajności oraz jak wzrost wydajności wynikający z tych ulepszeń może rosnąć wraz ze wzrostem rozmiaru nakładów wzrasta. Na koniec dowiedzieliśmy się, jak profilować i porównywać w celu ulepszenia naszych wdrożeń.