R DAY 2018: ESTRUTURAS DE DADOS DISTRIBUÍDAS EM R

C. Lente

2018-05-22

Índice

O que veremos hoje:

- Estruturas de dados distribuídas
  - Motivação
  - Visão geral
- O pacote `ddR`
  - Implementação
  - Vantagens e desvantagens
- Trabalhos futuros
  - Rcpp e companhia
  - ALTREP

Estruturas de Dados Distribuídas

Uma Breve Introdução

- Em torno dos anos 1980 o limite físico dos seus componentes passou a ser um motivo de preocupação
- Será que podemos aumentar o clock para sempre?
  - A luz no vácuo se propaga a aproximadamente $3x10^{8}$m/s
  - A memória do seu computador está a apenas 5cm da CPU
  - O tempo de ida e volta da memória é $1.7x10^{-10}$s, limitando o clock a no máximo 6Ghz
- Ao longo dos anos 1980 e 1990, passou-se a investir pesadamente em tecnologias multithread e multicore
- Cada núcleo pode processar mais de um fluxo de instruções ao mesmo tempo e cada processador pode ter mais de um núcleo na sua placa
- **Mas como eu posso aproveitar essas novas tecnologias?**

Um Mar De Opções

- Não é trivial fazer algoritmos e estruturas de dados tirarem vantagem desse novo tipo de infraestrutura
- Race conditions, deadlocks, starvation e seus amigos são o pavor de muitos alunos de graduação em computação até os dias de hoje
- Ascensão da nuvem: infraestrutura baseada em clusters de computadores que conseguem se comunicar e dividir tarefas entre si
- Diversas bibliotecas tentam auxiliar o programador a aproveitar todas essas tecnologias e abstrair um pouco da sua complexidade:
  - Soluções como `OpenMP` e `OpenACC` permitem a paralelização de código com diretivas de compilação
  - O `Spark` é um framework que facilita a distribuição de conjuntos de dados ao longo de clusters de computadores
- **É fácil usar essas tecnologias? Como eu faço um data.frame em C++?**

Você Quer Desempenho Máximo?

```
struct Sum : public Worker
{
   const RVector<double> input;

   double value;

   Sum(const NumericVector input) : input(input), value(0) {}
   Sum(const Sum& sum, Split) : input(sum.input), value(0) {}

   void operator()(std::size_t begin, std::size_t end) {
      value += std::accumulate(input.begin() + begin, input.begin() +
   }

   void join(const Sum& rhs) {
      value += rhs.value;
   }
};
```

- **Dá pra mudar de slide? Meus olhos estão doendo...**

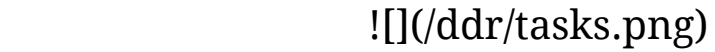
A Linguagem R

- Desde a sua concepção o R foi desenvolvido para funcionar com apenas uma thread
- Ao longo do tempo foram aparecendo diversas formas diferentes de trazer para o R as inovações do processamento paralelo
  - A Task View do CRAN sobre computação de alta performance (HPC) tem um total de *noventa e seis* pacotes
  - O cenário da HPC no R é fragmentado e pode não trazer o benefício de performance que o usuário espera

```
#> Unit: microseconds
#>                   expr      min   median       max
#>  mcmapply(print, 1:10) 6049.376 8889.710 10234.454
#>            print(1:10)  188.723  343.851   757.422
```

- **CVcê já está no 8º slide e não teve nenhum meme ainda!**

O R No Seu Computador



Estruturas De Dados

- A paralelização de uma analise de dados quase sempre envolverá a paralelização dos acessos a uma estrutura de dados (ED)
- Temos duas principais formas de paralelizar uma ED:
  - EDs concorrentes residem em memória compartilhada e podem ser acessadas por mais de um thread ao mesmo tempo
  - Se pudermos distribuir essas EDs ao longo de várias máquinas, teremos uma ED distribuída
- Para tarefas "padrão" de machine learning podemos usar o já citado `Spark`, que tem uma implementação própria de EDs distribuídas: Resilient Distributed Datasets (RDDs)
- A grande questão é que é difícil programar seus próprios algoritmos paralelos no `Spark`, acabamos dependendo do que já está implementado
- **C++ é impossível, Spark tem limitações... Como eu faço então?**

O Pacote ddR

Os Átomos Do Pacote

- A base do pacote são justamente as estruturas de dados; como discutido na seção anterior, elas são fundamentais para a paralelização
    - São 3 diferentes tipos de EDs (`darray`, `dframe` e `dlist`) e o `ddR` faz um bom trabalho de descrevê-las para nós

```
library(ddR)

as.dlist(1:10000)
#> ddR Distributed Object
#> Type: dlist
#> # of partitions: 10000
#> Partitions per
#> dimension: 10000x1
#> Partition sizes: [1], [1], [1], [1], [1], ...
#> Length: 10000
#> Backend: fork
```

- **Então essas EDs são concorrentes? Distribuídas? Os dois?**

Backends

- O `ddR` funciona com backends, permitindo que um mesmo código possa ser interpretado de forma concorrente ou distribuída
    - Por padrão o backend é o `fork`, mas com uma só linha podemos usar o `parallel` ou o `distributedR`

```
useBackend("parallel")

as.dlist(1:10000)
#> ddR Distributed Object
#> Type: dlist
#> # of partitions: 10000
#> Partitions per
#> dimension: 10000x1
#> Partition sizes: [1], [1], [1], [1], [1], ...
#> Length: 10000
#> Backend: parallel
```

- **Ok, Cuito bonito. Mas o que eu faço com isso?**

Um Exemplo

- A grande sacada do `ddR` é permitir que funções sejam aplicadas aos objetos através de uma *primitiva funcional* chamada `dmapply()`
    - A interface dessa função é igual ao `mapply()` mas ela tem completa integração com os backends, permitindo com que a aplicação seja concorrente ou distribuída

```
library(ddR)

f <- function(x) { length(runif(x)) }

microbenchmark::microbenchmark(
  base = mapply(f, 1:10000),
  ddR = dmapply(f, 1:10000))
#> Unit: milliseconds
#>  expr       min    median      max
#>  base 1402.0951 1429.4116 1467.857
#>   ddR  701.6403  721.4495  778.761
```

- **Isso é magia negra? Quero ver um outro exemplo!**

Outro Exemplo

```
means <- dmapply(mean, as.dlist(data.frame(1:4, 5:8, 9:12)))
means
# ddR Distributed Object
# Type: dlist
# # of partitions: 3
# Partitions per
# dimension: 3x1
# Partition sizes: [1], [1], [1]
# Length: 3
# Backend: parallel

collect(means)
# $X1.4
# [1] 2.5
#
# $X5.8
# [1] 6.5
#
# $X9.12
# [1] 10.5
```

Controle De Threds?

- A maior parte dos nossos problemas são vergonhosamente paralelizáveis, então podemos abrir mão do controle dos threads


- **Os cachorros são fofos, mas isso é tudo que o ddR faz?**

Algoritmos Embutidos

- O ddR já vem com alguns algoritmos pré-prontos para que você possa tirar vantagem das suas abstrações em alto nível

```
library(randomForest); library(randomForest.ddR)

microbenchmark::microbenchmark(
  rf = randomForest(medv ~ ., MASS::Boston),
  dfr = drandomForest(medv ~ ., MASS::Boston, nExecutor = 4))
#> Unit: milliseconds
#>  expr      min   median      max
#>    rf 594.8163 621.4649 899.2687
#>   drf 291.7429 309.5159 584.8597
```

- O ddR não é o framework mais veloz do mercado e não permite que o programador tenha controle fino sobre os threads de execução...
- *Mas* o que ele nos permite é ganhar muito desempenho sem ter que mudar nada da sintaxe que já conhecemos
- **Só acredito vendo... Sem nenhum gráfico você não me convence.**

Benchmarks

- Random Forest, K-Means e Regressão

Trabalhos Futuros

Novos Algoritmos

- O `ddR` já tem um punhado de algoritmos pré-prontos: `randomForest.ddR`, `pagerank.ddR`, `kmeans.ddR` e `glm.ddR`
- No meu trabalho de conclusão de curso, me propus a implementar mais dois algoritmos: `prcomp.ddR` e `gam.ddR`
- O primeiro destes está já foi implementado e está sendo otimizado
    - Para paralelizar o algoritmo da PCA, é necessário fazer uma estimação da matriz de variância-covariância a partir de cálculos realizados separadamente nas partições horizontais da tabela

$$n\mathbf{S} = \sum_{i=1}^{s} \mathbf{U}_i \mathbf{\Lambda}_i^2 \mathbf{U}_i^T + \sum_{i=1}^{s} n_i (\bar{\mathbf{x}}_i - \bar{\mathbf{x}})(\bar{\mathbf{x}}_i - \bar{\mathbf{x}})^T$$

- **Isso tudo é bem legal, mas e se eu quiser *mais* desempenho?**

C++

- Os desenvolvedores originais do pacote fizeram toda a sua infraestrutura em R, o que pode reduzir a performance da biblioteca
- Usando ferramentas como o `Rcpp`, o `RcppParallel` e o `RcppArmadillo` é possível otimizar o comportamento das funções do `ddR`, diminuindo o overhead de uma série de operações

```
microbenchmark::microbenchmark(
  otimizado = as_darray(matriz_1000_50, c(1, 50)),
  original = as.darray(matriz_1000_50, c(1, 50)))
#> Unit: microseconds
#>       expr        min      median         max neval
#>  otimizado    513.973    953.1695    8477.453   100
#>   original 204338.956 242818.3535 1492324.670   100
```

- No exemplo acima vemos que o processo de distribuição otimizado chega a ser ~250 vezes mais rápido do que o original
- **Existe alguma forma de isso tudo ser feito automaticamente pelo R?**

ALTREP

- `ALTREP` é um branch do do R que fornece um framework para o desenvolvimento de representações alternativas para objetos do R
    - Isso permite, por exemplo, o cálculo instantâneo da mediana de um vetor que tenha sido previamente ordenado
    - A `ALTVEC` (representação alternativa de vetores) já passou a ser suportada no R 3.5 lançado em 23/04/2018
- Se este framework começar a ser adotado no R padrão, será possível criar representações alternativas que por padrão já sejam distribuídas
- Teremos uma abstração completamente transparente ao usuário que poderá, por trás dos panos, otimizar computações que sejam identificadas como vergonhosamente paralelas
- **Mas isso é uma conversa para outra hora...**