iotools and ROctopus

Simon Urbanek

AT&T Labs - Research
Overview

• Big Data analytics in R
• High-throughput chunk-wise processing
  iotools + hmr
• In-memory computing
  ROctopus
• Lessons learned
• Conclusions
Big Data Analytics

• Data input/output
  - we typically do not have control over data formats
  - need efficient ways to convert on-disk data to native R objects

• R is great for sending computing to data
• Vectorized nature of R works well

• Two different approaches:
  - high throughput computing (typically one-time scan)
  - distributed in-memory computing (complex algorithms)
**iotools: high-performance I/O**

- originally created to optimize data loading: all raw input data lives in ASCII files
- chunk reader: content-aware reader of binary connections into raw vector buffers
- parses ASCII representation of data into R objects efficiently (uses raw vectors I/O, all parsing in-memory)
- uses matrix representation where possible (numeric or string), data frames otherwise
Parallelization via Map and Fold

\[ f(x) \downarrow \]

\[ f(x) = x^2 \]

\[ f'(x_1) \downarrow \quad f'(x_2) \downarrow \quad f'(x_3) \downarrow \]

\[ f''(f'(x_1), f'(x_2), f'(x_3)) \downarrow \]

\[ f''(x) = \text{concat} \]
Parallelization via Compute + Combine

\[ f(x) = \sum(x) \]

\[ f'(x) = \sum(x) \]

\[ f''(\ldots) = \sum(c(\ldots)) \]
Compute/combine processing

• At least three stages:
  - split (often implicit)
  - compute
  - combine

• Define functions using this paradigm
  simple examples:

```r
cc.sum <- function(x) cc(x, sum, sum)
cc.table <- function(x) cc(x, table, function(x) tapply(x, names(x), sum))
cc.mean <- function(x) cc(x, function(x) c(sum(x), length(x)),
                         function(x) sum(x[1,]) / sum(x[2,]))
```
**iotools: hmr() - Hadoop Map Reduce**

- iotools: highly efficient chunk-wise I/O on streams: let’s use it with Hadoop streaming!
- “formatters” define how to parse raw bytes into R objects - iotools provide ASCII-based formatters
- “as.output” method for arbitrary serialization, iotools provide by default to ASCII conversion
- handles matrices, vectors, tables, ... intuitively (row) names are treated as keys
- very efficient and R-native syntax
Example

• Aggregate point locations by ZIP code (match points against ZCTA US/Census 2010 shapefiles)

```r
r <- read.table(open(hmr(hinput("/data/2014/08"),
function(x)
  table(zcta2010.db()[
    inside(zcta2010.shp(), x[,4], x[,5]), 1]),
function(x) ctapply(as.numeric(x), names(x), sum)))
```

• Fairly native R programming

• Implicit defaults (read.table parser, conversion of named vectors to key/value entries)
hmr() flow

- **data**
  - chunk
  - chunk
  - chunk

- formatter
- matrix
- mapper
- name/values
  - (TAB separated ASCII)

- combine names
  - (Hadoop shuffle+sort)

- formatter
- matrix
- reducer
- name/values
  - HDFS

- as.output()
hmr behind the scenes

- loads the same packages as the calling session
- auxiliary data can be passed using `aux` parameter - it will be serialized and sent to the cluster (we deliberately make this explicit)
- indices, hash tables, etc. can be re-used across chunks (R is extremely efficient with hash lookup and joins - see also `fastmatch`, `mindex`, ...)
- reducers use key-aware chunking such that chunks will grow to contain all rows for a key
iotools vs classic map/reduce

```r
library(iotools)

hmr(hinput("twitter", formatter = function(x) mstrsplit(x, "\n", quiet = TRUE)),
    map = function(m) table(unlist(strsplit(m, "[^a-zA-Z]+"))),
    reduce = function(m) ctapply(as.numeric(m), names(m), sum))

library(rmr2)

mapreduce(
    input = "twitter", input.format = "text",
    map = function(., lines)
        keyval(unlist(strsplit(x = lines, split = "[^a-zA-Z]+")), 1),
    reduce = function(word, counts) keyval(word, sum(counts)),
    combine = TRUE)
```
Main differences

• chunk-wise processing
  - iotools read a chunk (typically dozens of megabytes) at a time - can contain many keys at once
  - one function evaluation per chunk, not key/value pair!
  - reducer chunking “smart” to not break keys apart
• highly efficient parsing
  - low-level optimization when creating R objects
• “native” R objects
  - more natural R programming: matrices, data.frames
Report from the (AT&T) trenches

- popular claim: M/R is not sufficient, not all algorithms fit
- true, BUT in real life raw data is big and needs heavy processing before one can even think about modeling (typical example: ca. 200Tb input)
- most models typically fit on today’s machines (mid-size machines have ca. 0.5Tb RAM, 40HTs)
- in the vast majority of applications M/R works just fine pre-processing and even model search
- for those few that don’t ...
ROctopus

• Using R containers as “hot” in-memory compute elements
• Native support for passing closures across R instances - send code to data
• Easy to send data/code from any container to another container
• Example configuration: use iotools/hmr to load data into ROctopus
ROctopus benefits

• no data transfer needed
  - transfer only update information where needed
• no data conversion after loading - the session is “hot” with R objects ready to compute on
• arbitrary additional state can be kept in the workspace for fast updates (abstraction for data + abstraction of mutable state)
• simple R function calls (very similar to snow!)
• each container has an address (URL) by which it can be addressed (any-to-any)
ROctopus example: GLM

dev = \texttt{wqapply}(d, \texttt{sum}(\texttt{fam$dev.resids(y, mu, WEIGHTS)}), \texttt{fold=`+`)}

\texttt{for} (\texttt{iter} \texttt{in} 1L:}\texttt{maxit}) \{
    \texttt{XtX} = \texttt{wqapply}(d, \\
    \texttt{crossprod( mm[, , drop = FALSE] * w ), fold=`+`})

    \texttt{Xty} = \texttt{wqapply}(d, \texttt{t(mm[, , drop = FALSE] * w)} \%\% \texttt{(z * w)}, \\
    \texttt{fold=`+`})

    \texttt{beta} = \texttt{solve(XtX, Xty)}

    \texttt{wrun}(d, \texttt{bquote(update_vals(.)(beta))})

    \texttt{devold} = \texttt{dev}
    \texttt{dev} = \texttt{wqapply}(d, \texttt{sum(\texttt{fam$dev.resids(y, mu, WEIGHTS)}), fold=`+`)}

    \texttt{cat("Deviance = ", dev, " Iterations - ", iter, "\n", sep = ",")}
    \texttt{if (abs(dev - devold)/(0.1 + abs(dev)) < epsilon) break;}
}\}
Models

• Generalized Linear Models
  - Logistic regression
  - Multinomial, mixed, ordered logit
  - Probit, multinomial and ordered probit
  - Poisson
  - Survival analysis

• Regularized LSQ (Ridge regression)

• Planned: LASSO

implemented by Mike Kane using foreach, moving to ROctopus
Lessons from ROctopus

• Low-level API
  - evaluate code in target instance (any-to-any)

• Mid-level API
  - common paradigms: e.g., distribute, evaluate, collect

• High-level API: include data abstraction
  - data represented as an opaque object
Lessons from iotools

- Work on native R objects simplifies debugging and development
- Leverage R’s strengths (vectorization, functional nature)
- “natural” R programming is very important for adoption
- R works well on Big Data when scaled
Desired Action Items

• Core frameworks (API)
  - for chunkwise processing (streaming)
  - for compute/combine (distributed HTP)

  such that packages can provide streaming and distributed algorithms

• Unification of distributed computing
  - merge concepts from snow and other packages in an implementation-independent manner
  - also define higher-level APIs
Conclusions

• Need for distributed API such that packages can provide algorithms *independently* of the back-end
• chunk/blockwise processing is a must for large scale analytics
• Path via compute/combine was very successful for practical use (iotools/hmr)
• In-memory computing is converging but its place is not quite settled. But we should have an R solution
Contact

• Simon Urbanek
  simon.urbanek@R-project.org

• iotools/hmr
  http://github.com/s-u/iotools
  http://github.com/s-u/hmr

• ROctopus
  (in progress, also on GitHub but don’t use yet ;))