Proposal for parallel sort in base R (and Python/Julia)

Directions in Statistical Computing

2 July 2016, Stanford

Matt Dowle

Initial timings

https://github.com/Rdatatable/data.table/wiki/Installation

See src/fsort.c

```
x = runif(N)
ans1 = base::sort(x, method='quick')
ans2 = data.table::fsort(x)
identical(ans1, ans2)
```

N=500m 3.8GB 8TH laptop: 65s => 3.9s (16x)

N=1bn 7.6GB 32TH server: 140s => 3.5s (40x)

N=10bn 76GB 32TH server: 25m => 48s (32x)

Reminder of problem dimensions …

1: “order” vs “sort”

“order” = find the order

- returns integer vector
- May be used many times downstream; e.g. data.table::setkey() uses it ncol(DT) times

- vs -

“sort” = sort the input

- Returns the input data sorted
- Possibly in-place

2: Stability

Stable

- Preserves the original appearance order of ties

- vs -

Unstable

- Doesn’t (usually unacceptable)

Not relevant for sort(), just order()

3: Cardinality

All unique

- runif(1e9)

- vs -

Duplicates (i.e. ties)

- sample(10, 1e9, replace=TRUE)

4: Range

range = [min(x), max(x)]

Small integer range => low cardinality

High integer range $\not\Rightarrow$ high cardinality

- x = c(1:1e4, 1e9)

5: Missingness

Are NA present at all?

- if not, can avoid deep branches

Do they come first or last?

- in data.table always first so user sees them

Are there a few NAs or mostly NAs?

- skew to one value but at least we know this value (NA) always sorts first or last

6: Types

logical

integer

bit64::integer64

double

character

factor

Each has a different strategy / optimization

7: Direction

Increasing

- vs -

Decreasing

- Should ties preserve original order or reverse order when decreasing?
- Efficiently switch direction without deep branches

8: Input Sortedness

- Already perfectly sorted?
  - short-circuit quickly

- Partially sorted?

  - minimize work

- Blocked?
  - Each duplicate is grouped together, but the groups are out of order
  - Move all items but in a batched fashion

- Thoroughly random?

9: Input Size

- Inputs less than 10MB fit in cache
  - all options are fast
- Divided input fits in cache
  - hybrid approaches
- Fastest for < 30 items is insert sort
- Fastest for 2 items is ?:

10: Multiple Columns

A list of N columns

Each a different type

Each column has *low* cardinality, typically

But combined *high* cardinality, typically

The ***order*** of the columns is significant

As per: data.table::setkey(DT, id, date)

11: Return groups?

Duplicates define groups

A by-product of sorting

Track the groups during sorting and then return them.

No more hash tables.

Works for high cardinality (small groups)

Detect full-cardinality (all unique) input and avoid returning N 1-item groups wastefully.

Efficient unique()

12: Skew

e.g. dividing into equal width bins won’t parallelize well if most values fall in a few bins due to skew

Hence nested parallelism? Potential thread management overhead.

Ideal to detect quickly the distribution and then switch to the most appropriate method.

13: Working Memory

- order usually uses more RAM than sort
  - sort can be in-place

- A single copy may not fit in RAM
  - not just speed but whether it works

14: Call Overhead

Iterating order() or sort() many times

- either internally or by users

Argument stack

Globals

Repeated memory allocation / GC

e.g. even memset() called many times unnecessarily can hurt performance

User API -vs- internal use

15: Multithreading

Thread safety of R

Don’t create a team of 32 threads to sort 2 numbers

Don’t create 1,000,000 threads

Do use 32 cores if you have 32 cores

Allow user to limit threads, though

Be “nice” to other process

Be “nice” to other users on the server

Follow CRAN policy: two threads

Stop on Ctrl-C

Load balance. Don’t have a slow or dead last thread.

Calling by users inside *their* parallel user code can bite

16: Specialization

Conceptually, for a vector x:

sort = x[order(x)]

Not as fast or memory efficient as a specialized :

sort(x)

Creating the order vector to use it and discard wastes time and RAM

Lazy evaluation and optimize as done by data.table within DT[...]

17: Code Complexity

Simpler code is better

- Easier to understand
- Easier to maintain
- Lower risk of bugs

Unless simpler code sucks at performance or results in out-of-memory

More complex code needs to be justified

18: User API

Progress bar

Verbose option to trace performance

Warnings

- “this double vector is really all integer”
- “these big ints are better as integer64”
- “btw, there’s a ton of 0.0 and -99.0”

19: Endianness

Little: Almost everything

Big: PowerPC and Solaris-Sparc

Sparc is proxy for PowerPC. We like and are thankful for CRAN's Sparc box. Some users do have big endian.

Currently, new radix order in base R is endian-aware. Would like to simplify and remove that.

20: Auto tuning

- Cache sizes vary; e.g. my laptop has 128MB L4 cache
- Cache configurations per socket vary
- CPU pipelines vary
- Compiler options vary

- Provide user API to determine optimal parameters for the hardware; e.g. when to switch between insert / counting / quick
  - tune_sort() => ~/.sortParams
- or be dynamic / use lscpu

What made it to base R last year?

Proposal at useR! 2015 Denmark

- It was order() not sort()
- Forwards radix
- All types, range > 100,000, double, character
- Returns grouping
- Partial sortedness detection
- High cardinality, small groups

Many thanks to Michael Lawrence for porting from data.table to base R

What am I proposing this year?

- Parallel sort() only
- Does not sort pieces then merge them
- Instead - radix count parallel histogram
- Currently just type double, >=0.0 and no NA
- Initial timings on slide 2 e.g. 25m => 48s

- Aside: for > 1bn, R's random number generator needs looking at. Use PCG rather than Mersenne Twister.

Your advice/guidance please

- What are existing solutions: STL, Python, Rth, Java8, TBB, Thrust, Boost, Spark ?
- In particular: any known non sort-merge parallel implementations?
- Benchmarking performance
- Correctness tests
- Literature review
- Porting to Python/Julia
- All 20 dimensions

And while I’m here …

data.table::fwrite

```
                                                   Laptop SSD             Server
                                                   4core/16gb       32core/256gb
                                                     10m rows          100m rows
                                                   ========== =================
                                                    Time Size RamDisk  HDD Size
                                                     Sec   GB    Time Time   GB
fwrite(DT,"fwrite.csv")                        csv     2  0.8       9   61  7.5
write_feather(DT, "feather.bin")               bin     5  1.0      27   75  9.1
save(DT,file="save1.Rdata",compress=F)         bin    11  1.2      90  137 12.0
save(DT,file="save2.Rdata",compress=T)         bin    70  0.4     647  679  2.8
write.csv(DT,"write.csv.csv",**)               csv    63  0.8     749  824  7.3
readr::write_csv(DT,"write_csv.csv")           csv   132  0.8    1997 1571  7.3

[**] row.names=F,quote=F
```

http://blog.h2o.ai/2016/04/fast-csv-writing-for-r/

Parallel subset

```
nrow(DT) == 200m
ncol(DT) == 4
object.size(DT) == 5GB
ix = sample(nrow(DT), nrow(DT)/2)
DT[ix]   # 20s => 3.5s with 16TH
```

Thanks to Arun for implementing parallel subset *within column*. So even a one column DT benefits too!

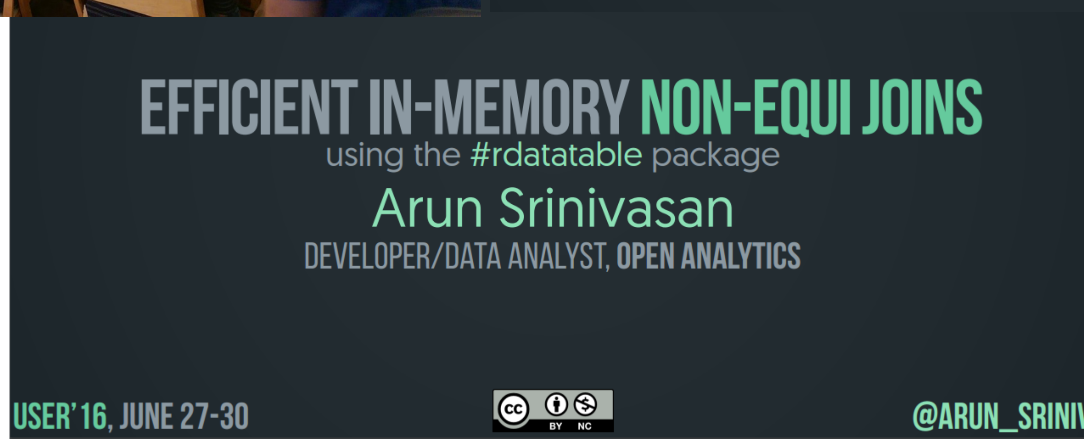
Non-equi joins

Presentation by Arun at useR! 2016 Stanford

PERFORMANCE

nrow(A) ~= 40m, nrow(B) ~= 33k		
Method	**Run Time(s)**	**Memory used (GB)**
dt-non-equi	4.9	1.2
dt-foverlaps	4.1	1.4
findOverlaps	6.2	2.1
RSQLite	87.0*	-

* nrow(A) = 100,000



Big join in H2O …

Ordered join like data.table

Parallel and distributed

Neither table need fit in one node's RAM

Very high cardinality

Here we test 200GB (10bn keys) joined to 200GB (10bn keys) returning 300GB (10bn keys)

Two table inputs

10bn rows
2 cols
200GB

```
$ head X
KEY,X2
2954985724,-92335012
5501052357,-8190789743
8723957901,-6631465068
706905226,-1289657629
706905226,7746956291
```

10bn rows
2 cols
200GB

```
$ head Y
KEY,Y2
706905226,3226855142
2954985724,-8875053263
3409724497,5353612273
8723957901,3462315357
2954985724,9186925123
```

Result
~10bn rows; 3 cols; 300GB

KEY	X2	Y2
706905226	-1289657629	3226855142
706905226	7746956291	3226855142
2954985724	-92335012	-8875053263
2954985724	-92335012	9186925123
8723957901	-6631465068	3462315357

Ordered by join column(s) for easier and faster subsequent operations

NB: Outer join is also implemented. Inner join is illustrated.

H2O commands are easy

```
library(h2o)
h2o.init(ip="mr-0xd6", port=55666)
X = h2o.importFile("hdfs://mr-
0xd6/datasets/mattd/X1e10_2c.csv")
Y = h2o.importFile("hdfs://mr-
0xd6/datasets/mattd/Y1e10_2c.csv")
ans = h2o.merge(X, Y, method="radix")
system.time(print(head(ans)))
```

H2O.ai
Machine Intelligence

Scaling

4 node

800GB/128cpu

1e6	6s
1e7	7s
1e8	13s
1e9	49s

10 node

2TB/320cpu

1e6	11s, 6s
1e7	6s
1e8	9s
1e9	30s
1e10	10m <= demo

H_2O.ai
Machine Intelligence

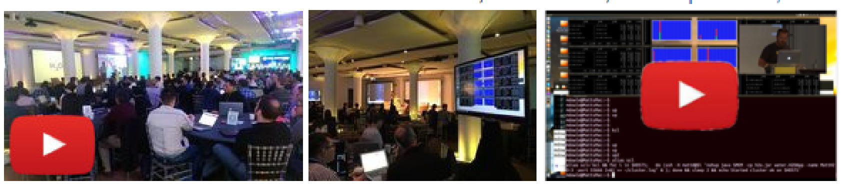
https://github.com/Rdatatable/data.table/wiki/Presentations

27 June 2016: Ninja Moves with data.table 3hr free tutorial, Matt Dowle & Arun Srinivasan, useR!2016, Stanford

---

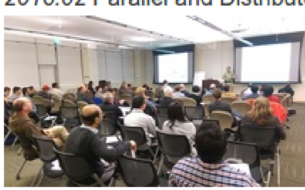
2016.05 : Parallel and Distributed Joins in H2O, Matt Dowle, Data by the Bay, San Francisco

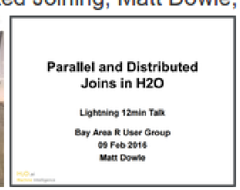
2016.05 : Parallel and Distributed Joins in H2O, Matt Dowle, H2O Open Tour, Chicago



2016.05 : R Lecture #3: data.table, Peter Hurford

2016.02 Parallel and Distributed Joining, Matt Dowle, Bay Area R User Group





DataCamp data.table

H_2O

Open Analytics data.table

Clone this wiki locally

https://github.com/Rdatat