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

Directions in Statistical Computing
2 July 2016, Stanford

Matt Dowle
Initial timings

See src/fsort.c

```r
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: Missingsness

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:
\[
\text{sort} = x[\text{order}(x)]
\]
Not as fast or memory efficient as a specialized:
\[
\text{sort}(x)
\]
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
Verbosed 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, $\geq 0.0$ and no NA
- Initial timings on slide 2 e.g. 25m => 48s

- Aside: for $> 1$bn, 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

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

\[ \text{nrow(DT)} == 200m \]
\[ \text{ncol(DT)} == 4 \]
\[ \text{object.size(DT)} == 5GB \]
\[ \text{ix} = \text{sample(nrow(DT)}, \text{nrow(DT)}/2) \]
\[ \text{DT[ix]} \quad \# \quad 20s \Rightarrow 3.5s \text{ with 16TH} \]

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**

\[
nrow(A) \sim 40m, \ nrow(B) \sim 33k
\]

<table>
<thead>
<tr>
<th>Method</th>
<th>Run Time(s)</th>
<th>Memory used (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>dt-non-equi</td>
<td>4.9</td>
<td>1.2</td>
</tr>
<tr>
<td>dt-foverlaps</td>
<td>4.1</td>
<td>1.4</td>
</tr>
<tr>
<td>findOverlaps</td>
<td>6.2</td>
<td>2.1</td>
</tr>
<tr>
<td>RSQlite</td>
<td>87.0*</td>
<td>-</td>
</tr>
</tbody>
</table>

\* 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

<table>
<thead>
<tr>
<th>$\text{head X}$</th>
<th>$\text{head Y}$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>KEY, X2</strong></td>
<td><strong>KEY, Y2</strong></td>
</tr>
<tr>
<td>$2954985724$, $-92335012$</td>
<td>$706905226$, $3226855142$</td>
</tr>
<tr>
<td>$5501052357$, $-8190789743$</td>
<td>$2954985724$, $-8875053263$</td>
</tr>
<tr>
<td>$8723957901$, $-6631465068$</td>
<td>$3409724497$, $5353612273$</td>
</tr>
<tr>
<td>$706905226$, $-1289657629$</td>
<td>$8723957901$, $3462315357$</td>
</tr>
<tr>
<td>$706905226$, $7746956291$</td>
<td>$2954985724$, $9186925123$</td>
</tr>
</tbody>
</table>
Result
~10bn rows; 3 cols; 300GB

<table>
<thead>
<tr>
<th>KEY</th>
<th>X2</th>
<th>Y2</th>
</tr>
</thead>
<tbody>
<tr>
<td>706905226</td>
<td>-1289657629</td>
<td>3226855142</td>
</tr>
<tr>
<td>706905226</td>
<td>7746956291</td>
<td>3226855142</td>
</tr>
<tr>
<td>2954985724</td>
<td>-92335012</td>
<td>-8875053263</td>
</tr>
<tr>
<td>2954985724</td>
<td>-92335012</td>
<td>9186925123</td>
</tr>
<tr>
<td>8723957901</td>
<td>-6631465068</td>
<td>3462315357</td>
</tr>
</tbody>
</table>

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

```r
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)))
```
## Scaling

<table>
<thead>
<tr>
<th></th>
<th>4 node</th>
<th>10 node</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>800GB/128cpu</td>
<td>2TB/320cpu</td>
</tr>
<tr>
<td>1e6</td>
<td>6s</td>
<td>1e6</td>
</tr>
<tr>
<td>1e7</td>
<td>7s</td>
<td>1e7</td>
</tr>
<tr>
<td>1e8</td>
<td>13s</td>
<td>1e8</td>
</tr>
<tr>
<td>1e9</td>
<td>49s</td>
<td>1e9</td>
</tr>
<tr>
<td>1e10</td>
<td>10m</td>
<td>1e10</td>
</tr>
</tbody>
</table>

"demo"
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

H2O.ai
Machine Intelligence