Scaling Data Analytics

Jan Vitek
Challenges

• How do we program big data?
• What are the tools?
• What are the abstractions?
• How do we debug, visualize, tune big data?
Some big data infrastructures

Hadoop

MapReduce

RHIPE

Pig

Hive

Flume/Java

X10
4 Myths

• Big data is big.
• Big data is speed.
• Big data is storage.
• Big data is hard.
Requirements

• Scale up vs. Scale down
• Rapid feedback, interaction with data, partial results
• Familiarity, ease of development
• Ease of deployment
• Portability and heterogeneity
• Robustness
• Efficiency
A tale of two communities

• **Computer Scientists**: Fixed programs, transient data.
  i.e. there will always be another input

• **Data Scientists**: Fixed data, transient programs.
  i.e. there will always be another query.

• *This dichotomy leads to a different world view in terms of design.*
  *In CS, languages/tools are built around static code abstractions. In DS, everything is dynamic and lightweight.*
High-level dynamic languages

• Programming is simplified by the language virtual machine
  • memory management
  • threading
  • platform heterogeneity
• At a cost
  • Performance
  • Footprint
ReactoR...

- ... create an open source platform for data analytics at scale
- ... built in collaboration by Purdue, INRIA, Stanford & Oracle
ReactoR Overview

- R+BigVector
- O2
- FastR

DS in R

- Java
- Substrate
- Hotspot
- LLVM

CS in R

Native Libraries

Oracle

OracleDB

NFS

Hadoop

Web

Oracle

Purdue

INRIA
Why R?

... language for data analysis and graphics

... open source

... books, conferences, user groups

... 4K+ packages

... 3mio users
Scripting data

read data into variables

make plots

compute summaries

more intricate modeling

develop simple functions
to automate analysis

...
Why Java?

... portable
... supports heterogenous platforms
... concurrent
... robust and stable
... fast enough
... books, conferences, user groups
... thousands of packages
... millions of developers
Scaling up…

Current limitations of R on a single node:

• Speed
• Memory footprint
• Limited support for concurrency
To understand where the time is spent, we turn to more representative R programs. Fig. 7 shows the breakdown of execution times in the Bioconductor dataset obtained with ProfileR. Each bar represents a Bioconductor vignette. The key observation is that memory management accounts for an average of 29% of execution time. Memory management time was further broken down into time spent in garbage collection (18.7%), allocating cons-pairs (3.6%), vectors (2.6%), and duplications (4%) for call-by-value semantics. The time spent in built-in functions represents the true computational work performed by R, this is on average 38% of execution time. There are some interesting outliers. The maximum spent in garbage collection is 70% and there is a program that spends 63% copying arguments. The lookup and match categories (4.3% and 1.8%) represent time spent looking up variables and matching parameters with arguments. Both of these would be absent in a more static language like C as they are resolved at compile time. Variable lookup will also be absent in Lisp or Scheme as, once bound, position of variables in a frame are known. Given the nature of R, many of the core numerical functions are written in C or Fortran. This can lead to the perception that execution time is dominated by native libraries. Looking at the amount of time spent in calls to foreign
To understand where the time is spent, we turn to more representative R programs. Fig. 7 shows the breakdown of execution times in the Bioconductor dataset obtained with ProfileR. Each bar represents a Bioconductor vignette. The key observation is that memory management accounts for an average of 29% of execution time. Memory management time was further broken down into time spent in garbage collection (18.7%), allocating cons-pairs (3.6%), vectors (2.6%), and duplications (4%) for call-by-value semantics. The time spent in built-in functions represents the true computational work performed by R, this is on average 38% of execution time. There are some interesting outliers. The maximum spent in garbage collection is 70% and there is a program that spends 63% copying arguments. The lookup and match categories (4.3% and 1.8%) represent time spent looking up variables and matching parameters with arguments. Both of these would be absent in a more static language like C as they are resolved at compile time. Variable lookup will also be absent in Lisp or Scheme as, once bound, position of variables in a frame are known. Given the nature of R, many of the core numerical functions are written in C or Fortran. This can lead to the perception that execution time is dominated by native libraries. Looking at the amount of time spent in calls to foreign...
Call-by-value semantics are implemented by a copy-on-write (COW) mechanism. Thus, the standard library only has three calls to a COW function. As mentioned above, they are really simple, on average only 37% of arguments are copied. It is interesting to note that the time spent testing to note that the time spent in function call arguments. It is interesting to note that the time spent in function call arguments. It is interesting to note that the time spent in function call arguments. It is interesting to note that the time spent in function call arguments.

Yet, 36% of vectors allocated in the CRAN code only eight, pairlists, 3.7% lists, and 46% other. The fact, the standard library only has three calls to a COW function. As mentioned above, they are really simple, on average only 37% of arguments are copied. It is interesting to note that the time spent in function call arguments. It is interesting to note that the time spent in function call arguments. It is interesting to note that the time spent in function call arguments.

Call-by-value semantics are implemented by a copy-on-write (COW) mechanism. Thus, the standard library only has three calls to a COW function. As mentioned above, they are really simple, on average only 37% of arguments are copied. It is interesting to note that the time spent in function call arguments. It is interesting to note that the time spent in function call arguments. It is interesting to note that the time spent in function call arguments.

Another reason for the large human-effort data is C. It also shows that, in many cases, the internal data required is more than the user data. Not only is R slow, but it also consumes significant amounts of memory. Unlike C

R is clearly slow and memory inefficient. Much more so than other dynamic languages. This is largely due to the combination of language features (call-by-value, extreme dynamism, lazy evaluation) and the lack of efficient built-in types.

Yet, 36% of vectors allocated in the CRAN code only eight, pairlists, 3.7% lists, and 46% other. The fact, the standard library only has three calls to a COW function. As mentioned above, they are really simple, on average only 37% of arguments are copied. It is interesting to note that the time spent in function call arguments. It is interesting to note that the time spent in function call arguments. It is interesting to note that the time spent in function call arguments.

Another reason for the large human-effort data is C. It also shows that, in many cases, the internal data required is more than the user data. Not only is R slow, but it also consumes significant amounts of memory. Unlike C

R is clearly slow and memory inefficient. Much more so than other dynamic languages. This is largely due to the combination of language features (call-by-value, extreme dynamism, lazy evaluation) and the lack of efficient built-in types.
FastR status

FastR is a new R virtual machine written in Java

- Aims for compatibility & completeness
- Abstract syntax tree interpreter (80% complete for core language)
- LLVM JIT compiler (30% complete)
- Substrate VM (10% complete)
Speedup of FASTR over GNU-R

Relative speedup (larger is better)

spectralnorm  fasta  nbody  fannkuch  binarytrees  mandelbrot  fastaredux  pidigits  regexdna
O2 is self-organizing computational cloud for analytics.

- Written in Java for portability and ease of deployment
- Provides BigVectors as arraylets that can be distributed, moved, and swapped to disk
- Provides a Distributed Fork/Join framework for both local and remote concurrent computation
Distributed F/J

for (int i : ntrees)
    trees[i] = new Tree(_data, maxDepth,...);
DRemoteTask.invokeAll(trees);
print("Trees done in " + timer);
## Single node Random Forest (O2 v Fortran/R)

<table>
<thead>
<tr>
<th>data</th>
<th>rows</th>
<th>size</th>
<th>avg tree sz</th>
<th>Tree build time</th>
</tr>
</thead>
<tbody>
<tr>
<td>iris</td>
<td>.15K</td>
<td>8KB</td>
<td>8</td>
<td>2ms 8ms</td>
</tr>
<tr>
<td>chess</td>
<td>196K</td>
<td>3.7MB</td>
<td>8</td>
<td>140ms 200ms</td>
</tr>
<tr>
<td>stego</td>
<td>7.5K</td>
<td>11MB</td>
<td>557</td>
<td>440ms 2.4s</td>
</tr>
<tr>
<td>kaggle/cs</td>
<td>100K</td>
<td>4.3MB</td>
<td>5321</td>
<td>420ms 1s</td>
</tr>
<tr>
<td>kaggle/as</td>
<td>580K</td>
<td>1.7GB</td>
<td>45894</td>
<td>-- 25s</td>
</tr>
<tr>
<td>covtype</td>
<td>8.7M</td>
<td>72MB</td>
<td>95393</td>
<td>-- 3s</td>
</tr>
</tbody>
</table>

Distributed random forest in 3K lines of Java on O2
Conclusions

• Scaling data analytics is about making it easier to turn idea into software

• It requires an integrated infrastructure that leverage advances in programming languages and compilers technology with a deep understanding of the domain.

• Interactive exploration and time to solution are the most important factors