The seven deadly sins of cloud computing research

Malte Schwarzkopf\textsuperscript{1}  Derek G. Murray\textsuperscript{2}  Steven Hand\textsuperscript{1}

\textsuperscript{1}University of Cambridge Computer Laboratory

\textsuperscript{2}Microsoft Research Silicon Valley
The **seven deadly sins**
of cloud computing research

**sin, n.** – common simplification or shortcut employed by researchers; may present threat to scientific integrity and practical applicability of research
The seven deadly sins of **cloud computing** research

*Large-scale data processing, cluster computing, scalable web application serving.*

**Not in focus:**

*Cloud storage, NoSQL databases, network protocols, cloud economics...*
Disclaimer #1

We have committed many of these sins ourselves in our own work.

Disclaimer #2

We are highlighting wide-spread issues here, not judging the value of specific research endeavours.
First sin:

Unnecessary distributed parallelism
Hey, my algorithm for processing lots of data is taking a long time!

You need to parallelize!
Really?

Parallelism is not free!
It always adds overheads...

Diminishing returns
We can only scale to a limit...
**Merge sort**

$O(n \log n)$ in serial case

Less is better

- **Serial**
- **Serial, external**
- **Local 4x parallel**
- **Local 4x parallel, external**
- **Distributed parallel**
Non-determinism
Races
Synchronization
Tedious debugging

%$§#$&! But parallelism is hard!
Messrs. Dean & Ghemawat
INTRODUCE THEIR AMAZING
MAP-REDUCE
(U.S. Pat. 7,650,331)

Stop worrying about:
- Synchronization
- Data motion
- Parallel coordination
- Failures
- Communication

TODAY!

Fits all parallelization problems!
A little bit of history...

"[...] the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time."

-- Dean et al., MapReduce paper, OSDI 2004

"Each machine had two 2GHz Intel Xeon processors with Hyper-Threading enabled, 4GB of memory, two 160GB IDE disks, and a gigabit Ethernet link."

-- Dean et al., MapReduce paper, OSDI 2004
But Big Data requires distributed parallelism! The data sets are just sooooo big!
Most “big data” isn’t that big...

Average input size observed in many production clusters is ~10–15 GB!
Second sin:
Assumption of performance homogeneity
"the cloud" is not homogeneous!

Schad et al., VLDB 2010       Li et al., IMC 2010
Barker et al., MMSys 2010    Wang et al., INFOCOM´10

Later in HotCloud:

EC2 instance heterogeneity  [Ou et al.]
Variance hurts predictability  [Bortnikov et al.]
Own results from 2010 follow...

... and by the way:

Neither are dedicated clusters!
EC2 performance variance

Disk read performance

100 EC2 m1.small instances

Mean and standard deviation over 9 samples
EC2 performance variance

Disk read

Disk write

Colours: different instances (randomly selected)
Values: means over 4 runs of bonnie++
Error bars: +/- 1σ (std. dev.) over 4 runs
EC2 performance variance

Values: means over 5 randomly selected instances
Error bars: +/- 1σ (std. dev.)
“The three R’s”

Describe and quantify performance variability
- Repeated runs (at least 5-10)

Rigor
Error bars and their meaning

Ensure the results are true across time and space
- Repeat runs at different times
- Different “hardware” (instance types, if EC2)

Repeatability
Sufficient information to repeat the experiment
- Hardware config / instance type
- Communication fabric / topology
- Dataset(s)

“three R’s” due to Vitek et al., R3 – Repeatability, Reproducibility and Rigor, SIGPLAN Notices, 2012
Third sin:
Picking the low-hanging fruit
<table>
<thead>
<tr>
<th>System</th>
<th>Speedup</th>
<th>Framework</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scarlett</td>
<td>20%</td>
<td>iMapReduce</td>
<td>5x</td>
</tr>
<tr>
<td>iHadoop</td>
<td>25%</td>
<td>Hyracks</td>
<td>16x</td>
</tr>
<tr>
<td>CIEL</td>
<td>50%</td>
<td>Hadoop++</td>
<td>20x</td>
</tr>
<tr>
<td>PACMan</td>
<td>50%</td>
<td>Spark</td>
<td>40x</td>
</tr>
<tr>
<td>HaLoop</td>
<td>85%</td>
<td>Prlter</td>
<td>50x</td>
</tr>
<tr>
<td>Mesos</td>
<td>2x</td>
<td>Incoop</td>
<td>80x</td>
</tr>
<tr>
<td>LATE scheduler</td>
<td>2x</td>
<td>CamDoop</td>
<td>180x</td>
</tr>
<tr>
<td>Sector/Sphere</td>
<td>2x</td>
<td>DVM</td>
<td>200x</td>
</tr>
<tr>
<td>Mantri</td>
<td>3x</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**HOT OFF THE PRESS!**

- Venkataraman et al.: 10x
- Ananthanarayanan et al.: 47%
How hard is it to beat Hadoop?

It depends!

Do all these optimizations compose?

Of course not!
Categories of optimizations

1. *in-memory (RAM) caching*

2. *memoization of results*

3. *exploitation of data locality*

4. *domain-specific algorithms*

5. *load vs. job runtime trade-off*
Number of tasks

Per-iteration time [sec]

Hadoop

CIEL

Number of tasks
"CIEL is faster than Hadoop [by 160s per iteration]"

"CIEL has less constant overhead than Hadoop and scales similarly well."
The graph shows the relationship between the number of input vectors and the time (in seconds) required for the k-means algorithm on a Hadoop platform. The time increases linearly with the number of input vectors, indicating a direct proportionality between the two variables.
Number of input vectors

Time [sec]

Hadoop
CIEL record I/O

Number of input vectors

k-means
The diagram shows the time in seconds (y-axis) as a function of the number of input vectors (x-axis) for different methods: Hadoop, CIEL record I/O, and MPI. The data points and trend lines indicate a linear relationship between the number of input vectors and the time taken, with Hadoop and CIEL record I/O having similar slopes but differing intercepts, and MPI being the most efficient in terms of time for the given number of input vectors.
Number of input vectors vs Time [sec]

- **Hadoop**
- **CIEL record I/O**
- **MPI**
- **CIEL array**

**k-means**

**Number of input vectors** (4M, 10M, 20M, 30M)
But MapReduce implementations are much cheaper in engineering time!

Quantify the performance sacrificed.
Actually, we may be okay...

Lots of new, domain-specific research systems improve on MapReduce.

Evaluate against the best-of-breed, not Hadoop/MapReduce!

Publish your code, if at all possible!
Here endeth the sermon 😊

(but there are four more sins!)
1. Unnecessary (distributed) parallelism
2. Assumption of resource homogeneity
3. Picking the low-hanging fruit
4. Forcing the abstraction
5. Unrepresentative clusters/workloads
6. Assumption of perfect elasticity
7. Ignorance towards fault tolerance

Now is the time to confess!
(or to shoot the messenger 😊 )
FURTHER SINS

[these were not part of the HotCloud 2012 presentation, but included in other versions of this talk]
Fourth sin:
Forcing the abstraction
"Java MapReduce [...] is] the assembly language of Apache Hadoop"

-- Cloudera executive, Stanford EE380 class

Flume

Pig Latin

Hive

Incremental processing

Oozie

Databases/Query langs

MapReduce
Eureka!
Everything is a MapReduce!

... really?!
Assembly languages...

... have small, fine-grained, fast, composable instructions.

MapReduce was designed for...

... long-running, large, coarse-grained, massively parallel workloads!
Fifth sin:
Use of unrepresentative workloads and clusters
The three cluster types

1. (usually) virtual
2. single-purpose, single-job
3. single-user
The three cluster types

1. (usually) virtual
   - single-purpose, single-job
   - single-user

2. physical/virtual
   - single-purpose, multi-job
   - multi-user
The three cluster types

1. (usually) virtual
   - single-purpose, single-job
   - single-user

2. physical/virtual
   - single-purpose, multi-job
   - multi-user

3. physical/virtual
   - multi-purpose, multi-job
   - multi-user
The three cluster types

1. (usually) virtual
   - single-purpose, single-job
   - single-user

2. physical/virtual
   - Often used by academics...
   - ...but not representative of type 3!
   - So don’t imply it is!

3. physical/virtual
   - multi-purpose, multi-job
   - multi-user
We need “cluster mix“ benchmarks!

Starting points:

Hadoop GridMix

Google Trace

Berkeley SPIM
Sixth sin:
Assumption of perfect elasticity
[...] using 1000 servers for one hour costs no more than using one server for 1000 hours.

Companies with large batch-oriented tasks can get results as quickly as their programs can scale, since using 1000 servers for one hour costs no more than using one server for 1000 hours.

Scalability and resources are finite in reality!

- Unexpected bottlenecks
- Scheduling load
- Communication structure
- Increased likelihood of failures
Seventh sin:
Ignoring fault tolerance
Fault tolerance is a 1st class feature in MapReduce...

[…] we provide a fault-tolerant implementation that scales to thousands of processors. In contrast, most [existing] parallel processing systems […] leave the details of handling machine failures to the programmer.

-- Dean et al., MapReduce paper, OSDI 2004

... but it is often treated as second class!

- Rarely evaluated comprehensively

- In-memory caching often not fault tolerant

- Tricky with iterative processing on top of MapReduce
Iterative MapReduce example
Iterative MapReduce example
Iterative MapReduce example
Why do we sin?

*Sometimes because we lack data & infrastructure!*

➡️ **Industry, please help us!**

- Workload traces
- Statements of real-world problems
- Access to hardware

*Sometimes because we are lazy or working last-minute for a deadline...*

➡️ **Reviewers and shepherds can enforce standards (or we agree not to sin! 😊)**
How can we repent?

We agree to avoid the sins and heed the “three R“

- Consciously design experiments
- Justify when sins are unavoidable or irrelevant
- Listen to reviewers & shepherds

Allow sins to be exposed

- Make source code available
- Support reproduction/validation efforts
1. Compare serial to parallel implementation
   - Derive maximum parallel speedup
   - Justify going distributed

2. Repeated runs!
   - Indicate performance variance
   - Clearly state parameters
   - EC2: ideally, multiple clusters, multiple times of day

3. Do not use speedup-over-Hadoop as headline result!
   - Compare to relevant optimized alternatives
   - Or quantify speedup over serial (1 worker)
   - Release your source code, so others can build upon it!
- Decide if MapReduce is the correct abstraction
- If not, but cheaper, quantify loss in performance

- Consider different job types, priorities and preemption!
- Benchmark cluster “job mixes”

- Clearly qualify the elasticity assumptions made
- Note (potential) scalability bottlenecks

- Clearly state fault tolerance requirements
- Clearly state characteristics and techniques used
- Evaluate them!
1. Unnecessary (distributed) parallelism
2. Assumption of resource homogeneity
3. Picking the low-hanging fruit
4. Forcing the abstraction
5. Unrepresentative clusters/workloads
6. Assumption of perfect elasticity
7. Ignorance towards fault tolerance

Now is the time to confess!
(or to shoot the messenger 😊)