The seven deadly sins of cloud computing research

Malte Schwarzkopf [†] Derek G. Murray [‡] Steven Hand [†]
[†]*University of Cambridge Computer Laboratory* [‡]*Microsoft Research Silicon Valley*

Abstract

Research into distributed parallelism on "the cloud" has surged lately. As the research agenda and methodology in this area are being established, we observe a tendency towards certain common simplifications and shortcuts employed by researchers, which we provocatively term "sins". We believe that these sins, in some cases, are threats to the scientific integrity and practical applicability of the research presented. In this paper, we identify and discuss seven "deadly sins" (many of which we have ourselves committed!), present evidence illustrating that they pose real problems, and discuss ways for the community to avoid them in the future.

Sin 1 Unnecessary distributed parallelism

Parallel computing is an old concept, but has recently become a hot topic again due to two factors: multi-core CPUs, and the need to process very large data sets. The latter in particular has led the cloud community to adopt distributed parallelism as a default, often based on models such as MapReduce [11]. Unlike existing HPC frameworks for parallel processing, MapReduce offers an appealingly simple interface, and manages many aspects of the parallel coordination—such as synchronization, data motion and communication—as part of framework code.

However, it does not always make sense to parallelize a computation. Firstly, doing so almost inevitably comes with an additional overhead, both in runtime performance and in engineering time. Secondly, although MapReduce and similar data-flow frameworks aim to migitate the engineering overhead, this transparency often results in programmers being ignorant of the potential performance impact. This may arise from communicating large amounts of intermediate data across machine boundaries, or from using algorithms which inherently involve large amounts of frequently mutated shared state.

Thus, when designing a parallel implementation, its performance should *always* be compared to an optimized serial implementation, even if only for a small input data set, in order to understand the overheads involved.

If we satisfy ourselves that parallel processing is indeed necessary or beneficial, it is also worth considering whether *distribution* over multiple machines is required. As Rowstron *et al.* recently pointed out, the rapid increase in RAM available in a single machine combined with large numbers of CPU cores per machine can make it economical and worthwhile to exploit local, rather than distributed, parallelism [36]. This observation is especially pertinent given that use of MapReduce for processing relatively small data sets appears to be commonplace [3]. Harnessing locally available parallelism has attractive benefits: it is possible to exploit shared memory and fast communication primitives; and data motion, which is often a bottleneck in distributed data processing [9], is avoided. It is also worth noting that the original MapReduce system was developed and used at Google before multi-core CPUs were widely used. Modern many-core machines, however, can easily apply 48 or more CPUs to processing a 100+ GB dataset entirely in memory, which already covers many practical use cases.

There are, of course, many situations that warrant distributed parallelism: some data sets are too large for a single machine, or suffer performance degradation due to contention on shared I/O resources when running locally. The precise cross-over points between serial, locally parallelized and distributed implementation performance depend on the exact workload, and it thus makes sense to establish the need for distributed parallelism on a case-by-case basis, rather than assume it by default.

Sin 2 Assuming performance homogeneity

Systems for cloud computing usually run on, and are evaluated on, clusters of physical or virtual machines, with production systems comprising as many as thou-

sands of machines [11, 19]. In academic research, rented clusters of virtual machines in "the cloud" are popular evaluation testbeds, since most academics do not have access to large physical clusters. However, it has been shown that virtualized cloud environments exhibit highly variable and sometimes unpredictable performance [23, 37].

Any results obtained from such environments should hence be treated with care: it is absolutely essential to base reported results on multiple benchmark runs, and to report the performance variance over both a reasonable number of runs and a reasonable time period. Surprisingly many publications omit either [7, 10, 18, 24, 43, 44], are inconsistent [18, 42], or do not explicitly qualify parameters, such as the number of repeats, whether mean or median values are used, or what error bars on plots mean.

Variance may also exist for reasons other than the underlying multi-tenant cloud fabric: since cloud computing systems operate at a high level of abstraction, and frequently ship large amounts of data over the network, they are particularly vulnerable to performance interference from external effects such as network congestion or OS-level contention on shared resources. Again, many publications that use local clusters neglect to report on variance, or the absence thereof [6, 13, 14, 16, 25].

Finally, variance can also be caused by the system itself. For example, data loaded into the Hadoop Distributed File System (HDFS) is randomly distributed across the cluster [39], leading to a variable distribution of blocks corresponding to any single file. This particular issue has been ameliorated in some recent systems by the use of two-random-choice load balancing [15, 28], but some imbalance remains, and other issues inherent to system construction may exist, making measurement of performance variance in experiments a necessity.

Sin 3 Picking the low-hanging fruit

The Hadoop ecosystem is a highly popular basis for research in cloud computing—partly due to its open source nature, partly because of its generality and widespread use in production environments, and partly because it can easily be used as a benchmark for comparative evaluation. It is the last quality, in particular, that has turned out to be rather tempting to many.

Plenty of research projects have found ways of achieving a speedup over Hadoop for different kinds of workload, as depicted in Figure 1, ranging from double-digit percentages to order-of-magnitude speedups [2, 4, 6, 7, 12–14, 16, 18, 24, 29, 41, 43–45]. While impressive, the sheer magnitude and abundance of these speedup results does raise the question of their significance. To put it differently, one might ask how hard it is to achieve a

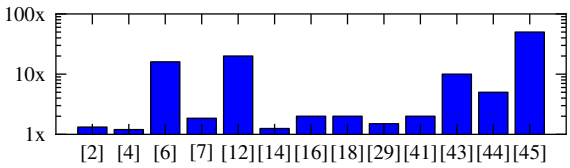
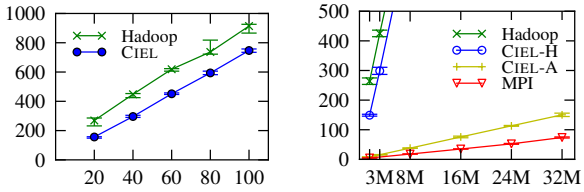


Figure 1: Maximum speedups over Hadoop claimed by a selection of research endeavours; N.B. log-scale y-axis.



(a) Tasks vs. per-iteration time [29]. (b) Input vectors vs. time [28].

Figure 2: k-means clustering with CIEL.

speedup over Hadoop? The answer is, of course, that it depends. Hadoop, as a consequence of its MapReduce roots, is an exceptionally general framework, while many of the research efforts mentioned are specialized to a particular application domain or class of algorithm. For example, iterative processing systems exploit in-memory caching [13, 34, 43, 45], and systems from the database community trade off additional processing at data load time for faster job execution times [1, 12].

It is unsurprising that it is easy to beat a general system, unoptimized for particular application requirements, by specializing it. Hence, a better comparison for any specialized system is with a *domain-specific optimized solution*, which may not be as scalable, fault-tolerant or easy-to-use as Hadoop, but provides a relevant performance baseline. Depending on the metric, we should not necessarily expect to beat the domain-specific solution—instead, the goal is to illustrate the *cost* of distributed processing, fault-tolerance or ease-of-use.

For example, we evaluated our CIEL system using the iterative *k*-means clustering algorithm. In our initial evaluation, we found CIEL to outperform Hadoop by about 160s per iteration using the same compute kernel [29] (Figure 2a). However, the margin is constant as the input data size and number of workers increase, meaning that CIEL has less constant overhead than Hadoop, but does not scale any better. Subsequently, we also built a more efficient *k*-means compute kernel using arrays instead of Hadoop-like record I/O, and compared its performance against an MPI-based implementation (see Figure 2b). We found that with this implementation, CIEL outperforms Hadoop by orders of magnitude, and scales better—but yet still performs, and scales, worse than MPI, which should be no surprise given its additional fault-tolerance overheads.

Furthermore, few researchers comment on the *composability* of their solutions. For example, if it were possible to compose all of the speedups depicted in Figure 1, a Hadoop job that previously took 24 hours to complete would be reduced to a duration of a mere 20 milliseconds! Clearly, some improvements proposed are very similar to, or encompass, others, while some might be mutually exclusive. If optimizations that have previously been published are used, this fact should be noted, and speedups should be reported relative to the existing work, rather than to "vanilla" Hadoop. PrIter [45] and Hadoop++ [12] are model citizens in this regard, quantifying the speedup over iMapReduce [44] and HadoopDB [1], respectively.

Sin 4 Forcing the abstraction

MapReduce is a versatile paradigm, covering a wide range of algorithms and problems, but it is not a panacea or silver bullet. Yet it is often used over other, more appropriate, approaches because it is simple and regarded as a de-facto "standard". As we note with Sin 3, the MapReduce paradigm has been applied to anything from databases to iterative processing.

As we consider this, it is worth recalling that MapReduce was designed with a primary design goal of scalably processing *huge* data sets in a *fault-tolerant* way. Indeed, one main design focus of MapReduce was to alleviate the I/O bottleneck of disk spindle performance by parallelizing over many disks [11], and dealing with the consequent additional failure scenarios. Furthermore, jobs were assumed to be processing so much data that they would take a long time, even when parallelized over many machines. Hadoop inherited this goal, and it is reflected in its design choices.[1]

There is, however, evidence that, at least in some places, many Hadoop jobs are quite short—median job lengths of around 90s have been reported [21, 43], although it is unclear whether these are representative of production environments. It does, however, seem to be the case that much research finding high speedups over Hadoop considers relatively small datasets (see Figure 3), which typically correspond to relatively small and short jobs.

This effect is compounded in the case of high-level languages that translate a program into a set of MapReduce jobs: Hive [40], Pig Latin [31] and FlumeJava [8] are recent examples. While we certainly agree that there is a case to be made for languages lending more expressivity to data-parallel paradigms, we believe that it is more appropriate to consider execution models that can perform iterations within a single job, instead of chaining MapReduce jobs, as this allows for whole program optimization and avoids having many short jobs.

[1]For example, task dispatch messages are piggy-backed onto Hadoop's periodic ping messages, so that a job can take up to 30s to start—this is insignificant for long-running jobs, but not for short ones.

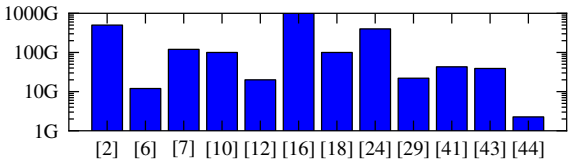


Figure 3: Max. input data set sizes considered in various research systems' evaluations; N.B. log-scale y-axis.

In some application domains, it is unclear if a MapReduce-like approach can offer any benefits, and indeed, some have argued that it is fruitless as a research direction [33].

In other application domains, the research community has a more pleasing track record of coming up with new, domain-specific systems. Examples of these include iterative processing [29, 43, 45], stream processing [17, 32], and graph processing [25, 26]. Ideally, future research in these application domains should build on these best-of-breed solutions, rather than on the MapReduce paradigm.

Sin 5 Unrepresentative workloads

Cloud computing workloads are evaluated on clusters of machines, and usually a particular type of cluster setup is implicitly assumed. Popular setups appear to be *(i)* large, shared, multi-framework, multi-job, multi-user clusters; *(ii)* single-purpose, but multi-user and multi-job clusters; or *(iii)* per-job (possibly virtual) clusters. There is evidence that some large data centre operators use the first option [27, 35, 38], and research on fair sharing of a cluster between multiple frameworks is emerging [18]. Furthermore, short batch computations can make up a large portion of a multi-job cluster's jobs, while using only a fraction of the resources [35]. Nonetheless, the common assumption in academic research systems is that the cluster workload is relatively *homogeneous*—that is, of the same job type and general nature—or that different kinds of workload do not interfere, or both. This assumption may stem from using workload traces with a single job type [3, 20, 43], or the assumption that in a "cloud" world of rented resources, clusters are spun up and down on demand. However, even in an environment such as EC2, interference exists, manifesting itself e.g. as contention on I/O resources such as disks and NICs, and resulting in variable performance (see Sin 2).

In a cluster shared between heterogeneous jobs, we may wish to prioritize some workloads over others. A common example of this is mixing revenue-critical web applications with backened data-analysis jobs, which are more tolerant of failures and extended runtimes [35].

This may necessitate preemptions of running tasks, which can also be used to improve locality [20].

Having established the widespread sharing of clusters between jobs in the "real world", we note that most research evaluations measure performance by running a single job on an otherwise idle cluster. This may be defended by noting that, at least with experiments on EC2, the effect of other jobs running manifests itself as performance variance (cf. Sin 2), but due to the random and uncontrollable nature of the variance, this affects the reproducibility of results, and it is unclear how closely it resembles the situation in large commercial clusters.

Instead, running a single job on an idle cluster is the moral equivalent of an OS micro-benchmark, and should be complemented by appropriate macro-benchmarks. In the OS community, the lack of a representative "desktop mix" benchmark prompted a call for better multi-core benchmarks [22], and in a similar vein, we believe the cloud computing community needs representative cluster benchmarks. Hadoop GridMix[2] is a good starting point, but only covers MapReduce jobs. Comprehensive cluster traces covering multiple job types, such as recent Google traces,[3] may aid generating synthetic workloads.

Sin 6 Assuming perfect elasticity

The cloud paradigm is closely associated with the concept of *utility computing* and its promise of an unlimited supply of computation. The notion that one can use 10 machines for 100 hours or 1000 machines for one hour at the same price [5], and get the same work done, is an oft-misquoted canard.

As researchers, we know that this is, of course, a fallacy, because workloads do not exhibit infinite parallel speedup. That said, even ignoring the issues of algorithmic scalability and financial concerns, the scalability and supply of compute resources are *not* in fact infinite. For example, as the size of a compute cluster grows, it runs increasingly many tasks, which are scheduled by a single cluster scheduler in contemporary architectures—clearly, scheduler throughput will eventually become a bottleneck. Similarly, there are limits to the scalability of data center communication infrastructure and logic. TCP incast is a well-known issue in large-scale distributed systems [30], and especially affects systems that operate a single "master" node, as is the case with most existing frameworks [11, 19, 29].[4] Another reason for diminishing returns from parallelization is the increasing likelihood of failures, and greater vulnerability to performance interference, resulting in, e.g. "straggler" tasks [41].

[2] `http://goo.gl/brd6y`

[3] `http://goo.gl/XO5YQ`

[4] Notably, Hadoop avoids this problem by only having workers communicate indirectly with the job tracker.

Sin 7 Ignoring fault tolerance

Two of the key challenges in the cloud are scale and fault tolerance. Indeed, the original MapReduce paper emphasizes ease-of-use and automatic handling of failures via task re-execution [11]. This is not a surprise: as jobs scale to thousands of tasks, the chance of the job being affected by a failure increases.

Yet despite this, many recent systems neglect to account for the performance implications of fault tolerance, or indeed of faults occurring [7, 10, 13]. This may be another reason for observing high speedups over Hadoop (cf. Sin 3): for example, a popular optimization is to store intermediate data in memory, but this obviously reduces fault tolerance and may necessitate re-running tasks.

Instead, for each system, we should ask the question whether fault tolerance is relevant or required for the application considered. If it is, it makes sense to check precisely what level is required, and what faults are to be protected against. The evaluation in this case should explicitly consider and ideally quantify the cost of fault tolerance. If it is not deemed to be required—e.g. because the job is running within a single fault tolerance domain—then this should, again, be argued explicitly.

Discussion and conclusions

We believe that the "sins" we discussed in this paper are widely committed—not maliciously, but for reasons of unwareness, or to simplify matters. While we do not believe that this (necessarily) invalidates existing research, we see a danger of these sins becoming entrenched in the research agenda and methodology.

Thus, we propose a set of remedies for the individual sins. Most of these can be best enforced by reviewers and shepherds, but we believe that, as a community, we should not require force—instead, we should cultivate awareness and avoid the sins by default, by following the suggestions below or otherwise.

1. Compare serial and distributed implementation performance of algorithms, or alternatively provide a formal or informal derivation of the maximum parallel speedup. Justify, if non-obvious, why the algorithm demands more parallelism than can be attained locally on a single machine.
2. Perform repeated runs and indicate performance variance on benchmarks, especially those executed in a virtualized environments. State clearly how many repeated runs the data is based on, and what the error bars signify (σ, max-min, percentiles, etc.).
3. Do not use or accept speedup over Hadoop as an indication of quality; instead, compare with relevant alternatives or justify why comparison with Hadoop is

appropriate. The parallel speedup over a serial execution is a good alternative metric, ideally compared to the maximum attainable speedup.

4. Use common sense, rather than ideology or ease-of-implementation, to decide if MapReduce is an appropriate paradigm for solving a problem. If it is not, but a MapReduce-based solution is significantly cheaper to implement, quantify the loss in performance by comparing with an appropriate alternative.
5. Consider different job types, priorities and preemption! Instead of, or in addition to, benchmarking individual jobs, benchmark cluster "job mixes".
6. When talking about elasticity, clearly qualify the assumptions made and note scalability bottlenecks.
7. Clearly state the fault tolerance characteristics of the proposed system, and justify why they are appropriate and satisfy the requirements.

We do realise that it may not be possible to avoid these sins all the time, but we believe that the *onus* should be on us as researchers to show that committing the sin is either unavoidable, or does not matter to answering the research question addressed. For some of the sins, such as the fifth and the sixth, this may be easy to argue; for others, such as the second, there really is no excuse ;-).

References

[1] ABOUZEID, A., ET AL. HadoopDB: An architectural hybrid of MapReduce and DBMS technologies for analytical workloads. *Proc. VLDB Endowment 2*, 1 (2009), 922–933.

[2] ANANTHANARAYANAN, G., ET AL. Reining in the outliers in map-reduce clusters using Mantri. In *Proc. OSDI 2010* (2010).

[3] ANANTHANARAYANAN, G., ET AL. Disk-Locality in Datacenter Computing Considered Irrelevant. In *Proc. HotOS* (2011), p. 1.

[4] ANANTHANARAYANAN, G., ET AL. Scarlett: coping with skewed content popularity in mapreduce clusters. In *Proc. EuroSys* (2011), pp. 287–300.

[5] ARMBRUST, M., FOX, A., ET AL. Above the clouds: a Berkeley view of cloud computing. Tech. rep., UC Berkeley, 2009.

[6] BORKAR, V., ET AL. Hyracks: A flexible and extensible foundation for data-intensive computing. In *Proc. ICDE* (2011), pp. 1151–1162.

[7] BU, Y., ET AL. HaLoop: Efficient iterative data processing on large clusters. *Proc. VLDB Endowment 3*, 1-2 (2010), 285–296.

[8] CHAMBERS, C., ET AL. FlumeJava: Easy, efficient data-parallel pipelines. In *Proc. PLDI* (2010), pp. 363–375.

[9] CHOWDHURY, M., ET AL. Managing data transfers in computer clusters with Orchestra. In *Proc. SIGCOMM* (2011), p. 98.

[10] CONDIE, T., ET AL. MapReduce online. In *Proc. NSDI* (2010), pp. 21–21.

[11] DEAN, J., ET AL. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. OSDI* (2004), p. 10.

[12] DITTRICH, J., ET AL. Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing). In *Proc. VLDB* (2010).

[13] EKANAYAKE, J., ET AL. Twister: a runtime for iterative mapreduce. In *Proc. HPDC* (2010), pp. 810–818.

[14] ELNIKETY, E., ET AL. iHadoop: asynchronous iterations for MapReduce. In *Proc. CloudCom* (2011), pp. 81–90.

[15] FETTERLY, D., ET AL. TidyFS: a simple and small distributed filesystem. In *Proc. USENIX ATC* (2011).

[16] GU, Y., ET AL. Sector and Sphere: the design and implementation of a high-performance data cloud. *Phil. trans. A, Math., phys., eng. sciences 367*, 1897 (2009), 2429–45.

[17] HE, B., ET AL. Comet: batched stream processing for data intensive distributed computing. In *Proc. SoCC* (2010), pp. 63–74.

[18] HINDMAN, B., ET AL. Mesos: A platform for fine-grained resource sharing in the data center. In *Proc. NSDI* (2011).

[19] ISARD, M., ET AL. Dryad: distributed data-parallel programs-from sequential building blocks. *SIGOPS OSR 41*, 3 (2007), 59.

[20] ISARD, M., ET AL. Quincy: fair scheduling for distributed computing clusters. In *Proc. SOSP* (2009), pp. 261–276.

[21] KAVULYA, S., ET AL. An Analysis of Traces from a Production MapReduce Cluster. In *Proc. CCGRID* (2010), pp. 94–103.

[22] KUZ, I., ET AL. Multicore OS benchmarks: we can do better. *Proc. HotOS* (2011).

[23] LI, A., ET AL. CloudCmp: comparing public cloud providers. In *Proc. IMC* (2010), pp. 1–14.

[24] LIU, H., ET AL. Cloud MapReduce: A MapReduce Implementation on Top of a Cloud Operating System. In *Proc. CCGRID* (2011), pp. 464–474.

[25] LOW, Y., ET AL. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *Proc. VLDB Endow. 5*, 8 (2012), 716–727.

[26] MALEWICZ, G., ET AL. Pregel: a system for large-scale graph processing. In *Proc. SIGMOD* (2010), pp. 135–146.

[27] MISHRA, A., ET AL. Towards characterizing cloud backend workloads: insights from Google compute clusters. *SIGMETRICS Perf. Eval. Review 37*, 4 (2010), 34–41.

[28] MURRAY, D. G. *A distributed execution engine supporting data-dependent control flow*. PhD thesis, Univ. of Cambridge, 2011.

[29] MURRAY, D. G., ET AL. CIEL: a universal execution engine for distributed data-flow computing. In *Proc. NSDI* (2011).

[30] NAGLE, D., ET AL. The Panasas ActiveScale storage cluster: Delivering scalable high bandwidth storage. In *Proc. SC* (2004).

[31] OLSTON, C., ET AL. Pig Latin: a not-so-foreign language for data processing. In *Proc. SIGMOD* (2008), pp. 1099–1110.

[32] OLSTON, C., ET AL. Nova: continuous Pig/Hadoop workflows. In *Proc. SIGMOD* (2011).

[33] PAVLO, A., ET AL. A comparison of approaches to large-scale data analysis. In *Proc. SIGMOD* (2009), pp. 165–178.

[34] POWER, R., ET AL. Piccolo: building fast, distributed programs with partitioned tables. In *Proc. OSDI* (2010).

[35] REISS, C., ET AL. Characterizing a large, heterogeneous cluster trace. Tech. rep., Carnegie Mellon University, Pittsburgh,, 2012.

[36] ROWSTRON, A., ET AL. Nobody ever got fired for using Hadoop. In *Proc. HotCDP* (2012).

[37] SCHAD, J., ET AL. Runtime measurements in the cloud: observing, analyzing, and reducing variance. *Proc. VLDB Endowment 3*, 1-2 (2010), 460–471.

[38] SHARMA, B., ET AL. Modeling and synthesizing task placement constraints in Google compute clusters. *Proc. SoCC* (2011).

[39] SHVACHKO, K., ET AL. The Hadoop distributed file system. In *Proc. IEEE MSST* (2010), pp. 1–10.

[40] THUSOO, A., ET AL. Hive – a petabyte scale data warehouse using Hadoop. In *Proc. ICDE* (2010), pp. 996–1005.

[41] ZAHARIA, M., ET AL. Improving MapReduce performance in heterogeneous environments. In *Proc. OSDI* (2008), pp. 29–42.

[42] ZAHARIA, M., ET AL. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proc. EuroSys* (2010), pp. 265–278.

[43] ZAHARIA, M., ET AL. Spark: Cluster computing with working sets. In *Proc. HotCloud* (2010), p. 10.

[44] ZHANG, Y., ET AL. iMapReduce: A Distributed Computing Framework for Iterative Computation. In *Proc. IPDPS* (2011).

[45] ZHANG, Y., ET AL. PrIter: a distributed framework for prioritized iterative computations. In *Proc. SOCC* (2011), ACM, p. 13.