Scalability! But at what COST?

Frank McSherry Michael Isard Derek G. Murray
Unaffiliated Unaffiliated* Unaffiliated†

Abstract

We offer a new metric for big data platforms, COST, or the Configuration that Outperforms a Single Thread. The COST of a given platform for a given problem is the hardware configuration required before the platform outperforms a competent single-threaded implementation. COST weighs a system's scalability against the overheads introduced by the system, and indicates the actual performance gains of the system, without rewarding systems that bring substantial but parallelizable overheads.

We survey measurements of data-parallel systems recently reported in SOSP and OSDI, and find that many systems have either a surprisingly large COST, often hundreds of cores, or simply underperform one thread for all of their reported configurations.

1 Introduction

> "You can have a second computer once you've shown you know how to use the first one."
>
> –Paul Barham

The published work on big data systems has fetishized scalability as the most important feature of a distributed data processing platform. While nearly all such publications detail their system's impressive scalability, few directly evaluate their absolute performance against reasonable benchmarks. To what degree are these systems truly improving performance, as opposed to parallelizing overheads that they themselves introduce?

Contrary to the common wisdom that effective scaling is evidence of solid systems building, any system can scale arbitrarily well with a sufficient *lack* of care in its implementation. The two scaling curves in Figure 1 present the scaling of a Naiad computation before (system A) and after (system B) a performance optimization is applied. The optimization, which removes parallelizable overheads, damages the apparent scalability despite resulting in improved performance in all configurations.

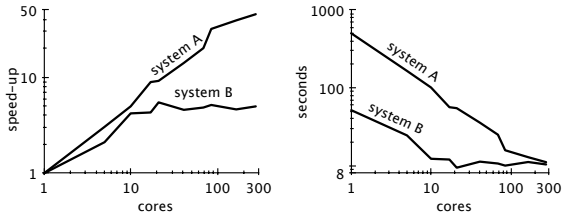

Figure 1: Scaling and performance measurements for a data-parallel algorithm, before (system A) and after (system B) a simple performance optimization. The unoptimized implementation "scales" far better, despite (or rather, because of) its poor performance.

While this may appear to be a contrived example, we will argue that many published big data systems more closely resemble system A than they resemble system B.

1.1 Methodology

In this paper we take several recent graph processing papers from the systems literature and compare their reported performance against simple, single-threaded implementations on the same datasets using a high-end 2014 laptop. Perhaps surprisingly, many published systems have *unbounded* COST—i.e., no configuration outperforms the best single-threaded implementation—for all of the problems to which they have been applied.

The comparisons are neither perfect nor always fair, but the conclusions are sufficiently dramatic that some concern must be raised. In some cases the single-threaded implementations are more than an order of magnitude faster than published results for systems using hundreds of cores. We identify reasons for these gaps: some are intrinsic to the domain, some are entirely avoidable, and others are good subjects for further research.

We stress that these problems lie not necessarily with the systems themselves, which may be improved with time, but rather with the measurements that the authors provide and the standard that reviewers and readers demand. Our hope is to shed light on this issue so that future research is directed toward distributed systems whose scalability comes from advances in system design rather than poor baselines and low expectations.

*Michael Isard was employed by Microsoft Research at the time of his involvement, but is now unaffiliated.

†Derek G. Murray was unaffiliated at the time of his involvement, but is now employed by Google Inc.

name	twitter_rv [13]	uk-2007-05 [5, 6]
nodes	41,652,230	105,896,555
edges	1,468,365,182	3,738,733,648
size	5.76GB	14.72GB

Table 1: The "twitter_rv" and "uk-2007-05" graphs.

```
fn PageRank20(graph: GraphIterator, alpha: f32) {
  let mut a = vec![0f32; graph.nodes()];
  let mut b = vec![0f32; graph.nodes()];
  let mut d = vec![0f32; graph.nodes()];

  graph.map_edges(|x, y| { d[x] += 1; });

  for iter in 0..20 {
    for i in 0..graph.nodes() {
      b[i] = alpha * a[i] / d[i];
      a[i] = 1f32 - alpha;
    }

    graph.map_edges(|x, y| { a[y] += b[x]; });
  }
}
```

Figure 2: Twenty PageRank iterations.

2 Basic Graph Computations

Graph computation has featured prominently in recent SOSP and OSDI conferences, and represents one of the simplest classes of data-parallel computation that is not trivially parallelized. Conveniently, Gonzalez et al. [10] evaluated the latest versions of several graph-processing systems in 2014. We implement each of their tasks using single-threaded C# code, and evaluate the implementations on the same datasets they use (see Table 1).[1]

Our single-threaded implementations use a simple Boost-like graph traversal pattern. A `GraphIterator` type accepts actions on edges, and maps the action across all graph edges. The implementation uses unbuffered IO to read binary edge data from SSD and maintains per-node state in memory backed by large pages (2MB).

2.1 PageRank

PageRank is an computation on directed graphs which iteratively updates a rank maintained for each vertex [19]. In each iteration a vertex's rank is uniformly divided among its outgoing neighbors, and then set to be the accumulation of scaled rank from incoming neighbors. A dampening factor `alpha` is applied to the ranks, the lost rank distributed uniformly among all nodes. Figure 2 presents code for twenty PageRank iterations.

[1]Our C# implementations required some manual in-lining, and are less terse than our Rust implementations. In the interest of clarity, we present the latter in this paper. Both versions of the code produce comparable results, and will be made available online.

scalable system	cores	twitter	uk-2007-05
GraphChi [12]	2	3160s	6972s
Stratosphere [8]	16	2250s	-
X-Stream [21]	16	1488s	-
Spark [10]	128	857s	1759s
Giraph [10]	128	596s	1235s
GraphLab [10]	128	249s	833s
GraphX [10]	128	419s	462s
Single thread (SSD)	1	300s	651s
Single thread (RAM)	1	275s	-

Table 2: Reported elapsed times for 20 PageRank iterations, compared with measured times for single-threaded implementations from SSD and from RAM. GraphChi and X-Stream report times for 5 PageRank iterations, which we multiplied by four.

```
fn LabelPropagation(graph: GraphIterator) {
  let mut label = (0..graph.nodes()).to_vec();
  let mut done = false;

  while !done {
    done = true;
    graph.map_edges(|x, y| {
      if label[x] != label[y] {
        done = false;
        label[x] = min(label[x], label[y]);
        label[y] = min(label[x], label[y]);
      }
    });
  }
}
```

Figure 3: Label propagation.

Table 2 compares the reported times from several systems against a single-threaded implementations of PageRank, reading the data either from SSD or from RAM. Other than GraphChi and X-Stream, which re-read edge data from disk, all systems partition the graph data among machines and load it in to memory. Other than GraphLab and GraphX, systems partition edges by source vertex; GraphLab and GraphX use more sophisticated partitioning schemes to reduce communication.

No scalable system in Table 2 consistently outperforms a single thread, even when the single thread repeatedly re-reads the data from external storage. Only GraphLab and GraphX outperform *any* single-threaded executions, although we will see in Section 3.1 that the single-threaded implementation outperforms these systems once it re-orders edges in a manner akin to the partitioning schemes these systems use.

2.2 Connected Components

The connected components of an undirected graph are disjoint sets of vertices such that all vertices within a set

scalable system	cores	twitter	uk-2007-05
Stratosphere [8]	16	950s	-
X-Stream [21]	16	1159s	-
Spark [10]	128	1784s	$\geq$ 8000s
Giraph [10]	128	200s	$\geq$ 8000s
GraphLab [10]	128	242s	714s
GraphX [10]	128	251s	800s
Single thread (SSD)	1	153s	417s

Table 3: Reported elapsed times for label propagation, compared with measured times for single-threaded label propagation from SSD.

scalable system	cores	twitter	uk-2007-05
GraphLab	128	249s	833s
GraphX	128	419s	462s
Vertex order (SSD)	1	300s	651s
Vertex order (RAM)	1	275s	-
Hilbert order (SSD)	1	242s	256s
Hilbert order (RAM)	1	110s	-

Table 4: Reported elapsed times for 20 PageRank iterations, compared with measured times for single-threaded implementations from SSD and from RAM. The single-threaded times use identical algorithms, but with different edge orders.

are mutually reachable from each other.

In the distributed setting, the most common algorithm for computing connectivity is label propagation [11] (Figure 3). In label propagation, each vertex maintains a label (initially its own ID), and iteratively updates its label to be the minimum of all its neighbors' labels and its current label. The process propagates the smallest label in each component to all vertices in the component, and the iteration converges once this happens in every component. The updates are commutative and associative, and consequently admit a scalable implementation [7].

Table 3 compares the reported running times of label propagation on several data-parallel systems with a single-threaded implementation reading from SSD. Despite using orders of magnitude less hardware, single-threaded label propagation is significantly faster than any system above.

3 Better Baselines

The single-threaded implementations we have presented were chosen to be the simplest, most direct implementations we could think of. There are several standard ways to improve them, yielding single-threaded implementations which strictly dominate the reported performance of the systems we have considered, in some cases by an additional order of magnitude.

3.1 Improving graph layout

Our single-threaded algorithms take as inputs edge iterators, and while they have no requirements on the order in which edges are presented, the order does affect performance. Up to this point, our single-threaded implementations have enumerated edges in *vertex order*, whereby all edges for one vertex are presented before moving on to the next vertex. Both GraphLab and GraphX instead partition the edges among workers, without requiring that all edges from a single vertex belong to the same worker, which enables those systems to exchange less data [9, 10].

A single-threaded graph algorithm does not perform explicit communication, but edge ordering can have a pronounced effect on the cache behavior. For example, the edge ordering described by a Hilbert curve [2], akin to ordering edges (a,b) by the interleaving of the bits of a and b, exhibits locality in both a and b rather than just a as in the vertex ordering. Table 4 compares the running times of single-threaded PageRank with edges presented in Hilbert curve order against other implementations, where we see that it improves over all of them.

Converting the graph data to a Hilbert curve order is an additional cost in pre-processing the graph. The process amounts to transforming pairs of node identifiers (edges) into an integer of twice as many bits, sorting these values, and then transforming back to pairs of node identifiers. Our implementation transforms the twitter_rv graph in 179 seconds using one thread, which can be a performance win even if pre-processing is counted against the running time.

3.2 Improving algorithms

The problem of properly choosing a good algorithm lies at the heart of computer science. The label propagation algorithm is used for graph connectivity not because it is a good algorithm, but because it fits within the "think like a vertex" computational model [15], whose implementations scale well. Unfortunately, in this case (and many others) the appealing scaling properties are largely due to the algorithm's sub-optimality; label propagation simply does more work than better algorithms.

Consider the algorithmic alternative of Union-Find with weighted union [3], a simple $O(m \log n)$ algorithm which scans the graph edges once and maintains two integers for each graph vertex, as presented in Figure 4. Table 5 reports its performance compared with imple-

scalable system	cores	twitter	uk-2007-05
GraphLab	128	242s	714s
GraphX	128	251s	800s
Single thread (SSD)	1	153s	417s
Union-Find (SSD)	1	15s	30s

Table 5: Times for various connectivity algorithms.

```
fn UnionFind(graph: GraphIterator) {
  let mut root = (0..graph.nodes()).to_vec();
  let mut rank = [0u8; graph.nodes()];

  graph.map_edges(|mut x, mut y| {
    while (x != root[x]) { x = root[x]; }
    while (y != root[y]) { y = root[y]; }
    if x != y {
      match rank[x].cmp(&rank[y]) {
        Less    => { root[x] = y; },
        Greater => { root[y] = x; },
        Equal   => { root[y] = x; rank[x] += 1; },
      }
    }
  });
}
```

Figure 4: Union-Find with weighted union.

mentations of label propagation, faster than the fastest of them (the single-threaded implementation) by over an order of magnitude.

There are many other efficient algorithms for computing graph connectivity, several of which are parallelizable despite not fitting in the "think like a vertex" model. While some of these algorithms may not be the best fit for a given distributed system, they are still legitimate alternatives that must be considered.

4 Applying COST to prior work

Having developed single-threaded implementations, we now have a basis for evaluating the COST of systems. As an exercise, we retrospectively apply these baselines to the published numbers for existing scalable systems.

4.1 PageRank

Figure 5 presents published scaling information from PowerGraph [9], GraphX [10], and Naiad [16], as well as two single-threaded measurements as horizontal lines. The intersection with the upper line indicates the point at which the system out-performs a simple resource-constrained implementation, and is a suitable baseline for systems with similar limitations (e.g., GraphChi and X-Stream). The intersection with the lower line indicates the point at which the system out-performs a feature-rich implementation, including pre-processing and sufficient

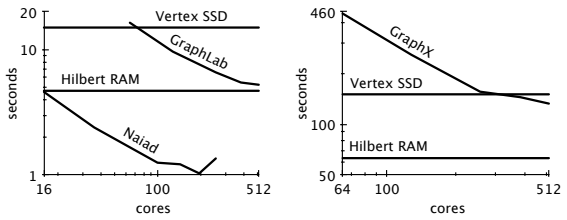

Figure 5: Published scaling measurements for PageRank on twitter_rv. The first plot is the time per warm iteration. The second plot is the time for ten iterations from a cold start. Horizontal lines are single-threaded measurements.

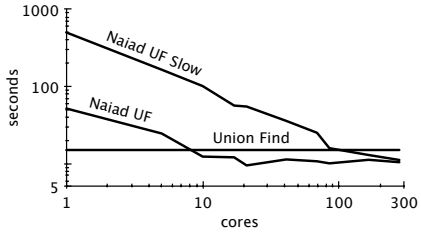

Figure 6: Two Naiad implementations of union find.

memory, and is a suitable baseline for systems with similar resources (e.g., GraphLab, Naiad, and GraphX).

From these curves we would say that Naiad has a COST of 16 cores for PageRanking the twitter_rv graph. Although not presented as part of their scaling data, GraphLab reports a 3.6*s* measurement on 512 cores, and achieves a COST of 512 cores. GraphX does not intersect the corresponding single-threaded measurement, and we would say it has unbounded COST.

4.2 Graph connectivity

The published works do not have scaling information for graph connectivity, but given the absolute performance of label propagation on the scalable systems relative to single-threaded union-find we are not optimistic that such scaling data would have lead to a bounded COST.

Instead, Figure 6 presents the scaling of two Naiad implementations of parallel union-find [14], the same examples from Figure 1. The two implementations differ in their storage of per-vertex state: the slower one uses hash tables where the faster one uses arrays. The faster implementation has a COST of 10 cores, while the slower implementation has a COST of roughly 100 cores.

The use of hash tables is the root cause of the factor of ten increase in COST, but it does provide some value: node identifiers need not lie in a compact set of integers. This evaluation makes the trade-off clearer to both system implementors and potential users.

5 Lessons learned

Several aspects of scalable systems design and implementation contribute to overheads and increased COST. The computational model presented by the system restricts the programs one may express. The target hardware may reflect different trade-offs, perhaps favoring capacity and throughput over high clock frequency. Finally, the implementation of the system may add overheads a single thread doesn't require. Understanding each of these overheads is an important part of assessing the capabilities and contributions of a scalable system.

To achieve scalable parallelism, big data systems restrict programs to models in which the parallelism is evident. These models may not align with the intent of the programmer, or the most efficient parallel implementations for the problem at hand. Map-Reduce intentionally precludes memory-resident state in the interest of scalability, leading to substantial overhead for algorithms that would benefit from it. Pregel's "think like a vertex" model requires a graph computation to be cast as an iterated local computation at each graph vertex as a function of the state of its neighbors, which captures only a limited subset of efficient graph algorithms. Neither of these designs are the "wrong choice", but it is important to distinguish "scalability" from "efficient use of resources".

The cluster computing environment is different from the environment of a laptop. The former often values high capacity and throughput over latency, with slower cores, storage, and memory. The laptop now embodies the personal computer, with lower capacity but faster cores, storage, and memory. While scalable systems are often a good match to cluster resources, it is important to consider alternative hardware for peak performance.

Finally, the implementation of the system may introduce overheads that conceal the performance benefits of a scalable system. High-level languages may facilitate development, but they can introduce performance issues (garbage collection, bounds checks, memory copies). It is especially common in a research setting to evaluate a new idea with partial or primitive implementations of other parts of the system (serialization, memory management, networking), asserting that existing techniques will improve the performance. While many of these issues might be improved with engineering effort that does not otherwise advance research, nonetheless it can be very difficult to assess whether the benefits the system claims will still manifest once the fat is removed.

There are many good reasons why a system might have a high COST when compared with the fastest purpose-built single-threaded implementation. The system may target a different set of problems, be suited for a different deployment, or be a prototype designed to assess components of a full system. The system may also provide other qualitative advantages, including integration with an existing ecosystem, high availability, or security, that a simpler solution cannot provide. As Section 4 demonstrates, it is nonetheless important to evaluate the COST, both to explain whether a high COST is intrinsic to the proposed system, and because it can highlight avoidable inefficiencies and thereby lead to performance improvements for the system.

6 Future directions (for the area)

While this note may appear critical of research in distributed systems, we believe there is still good work to do, and our goal is to provide a framework for measuring and making the best forward progress.

There are numerous examples of scalable algorithms and computational models; one only needs to look back to the parallel computing research of decades past. Borůvka's algorithm [1] is nearly ninety years old, parallelizes cleanly, and solves a more general problem than label propagation. The Bulk Synchronous Parallel model [24] is surprisingly more general than most related work sections would have you believe. These algorithms and models are richly detailed, analyzed, and in many cases already implemented.

Many examples of performant scalable systems exist. Both Galois [17] and Ligra [23] are shared-memory systems that significantly out-perform their distributed peers when run on single machines. Naiad [16] introduces a new general purpose dataflow model, and out-performs even specialized systems. Understanding what these systems did right and how to improve them is more important than re-hashing existing ideas in new domains compared against only the poorest of prior work.

We are now starting to see performance studies of the current crop of scalable systems [18], challenging some conventional wisdom underlying their design principles. Similar such studies have come from previous generations of systems [22], including work explicitly critical of the absolute performance of scalable systems as compared with simpler solutions [20, 4, 25]. While it is surely valuable to understand and learn from the performance of popular scalable systems, we might also learn that we keep making, and publishing, the same mistakes.

Fundamentally, a part of good research is making sure we are asking the right questions. "Can systems be made to scale well?" is trivially answered (in the introduction) and is not itself the right question. There is a substantial amount of good research to do, but identifying *progress* requires being more upfront about existing alternatives. The COST of a scalable system uses the simplest of alternatives, but is an important part of understanding and articulating progress made by research on these systems.

References

[1] http://en.wikipedia.org/wiki/Boruvka's_algorithm

[2] http://en.wikipedia.org/wiki/Hilbert_curve

[3] http://en.wikipedia.org/wiki/union_find

[4] Eric Anderson and Joseph Tucek, *Efficiency Matters!* HotStorage 2009.

[5] Paolo Boldi and Sebastiano Vigna, *The WebGraph Framework I: Compression Techniques*, WWW 2004.

[6] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. *UbiCrawler: A Scalable Fully Distributed Web Crawler* Software: Practice & Experience 2004.

[7] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohnler. *The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors*. SOSP 2013.

[8] Stephan Ewen, Moritz Kaufmann, Kostas Tzoumas, and Volker Markl. *Spinning Fast Iterative Data Flows*. VLDB 2012.

[9] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, Carlos Guestrin. *PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs*. OSDI 2012.

[10] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, and Michael J. Franklin, and Ion Stoica. *GraphX: Graph Processing in a Distributed Dataflow Framework*. OSDI 2014.

[11] U Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. *PEGASUS: Mining Peta-Scale Graphs*. ICDM 2009.

[12] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. *GraphChi: Large-Scale Graph Computation on Just a PC*. OSDI 2012.

[13] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. *What is Twitter, a social network or a news media?* WWW 2010.

[14] Silvio Lattanzi, Benjamin Moseley, Siddharth Suri, and Sergei Vassilvitskii. *Filtering: A Method for Solving Graph Problems in MapReduce*. SPAA 2011.

[15] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. *Pregel: A System for Large-Scale Graph Processing*. SIGMOD 2010.

[16] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. *Naiad: A Timely Dataflow System*. SOSP 2013.

[17] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. *A Lightweight Infrastructure for Graph Analytics*. SOSP 2013.

[18] Kay Ousterhout, Ryan Rasti?, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. *Making Sense of Performance in Data Analytics Frameworks*. NSDI 2015.

[19] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. *The PageRank citation ranking: Bringing order to the web*. Technical report, Stanford Digital Library Technologies Project, 1998.

[20] Alexander Rasmussen, George Porter, Michael Conley, Harsha V. Madhyastha Radhika Niranjan Mysore, Alexander Pucher, and Amin Vahdat. *TritonSort: A Balanced Large-Scale Sorting System*. NSDI 2011.

[21] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. *X-Stream: Edge-Centric Graph Processing using Streaming Partitions*. SOSP 2013.

[22] Mehul A. Shah, Samuel Madden, Michael J. Franklin, and Joseph M. Hellerstein. *Java Support for Data-Intensive Systems: Experiences Building the Telegraph Dataflow System*. SIGMOD Record 2001.

[23] Julian Shun and Guy Blelloch. *Ligra: A Lightweight Graph Processing Framework for Shared Memory*. PPoPP 2013.

[24] Leslie G. Valiant. *A bridging model for parallel computation*. Communications of the ACM, Volume 33 Issue 8, Aug. 1990.

[25] Ce Zhang and Christopher Ré. *DimmWitted: A Study of Main-Memory Statistical Analytics*. VLDB 2014.