

Scalability! But at what COST?

Jan 15, 2015

Michael Isard, Derek Murray, and I recently sent in a [HotOS](#) submission (it's not blind, so no harm talking about it, we think). The subject is hinted at from the post title (stolen from the paper title):

Big data systems may scale well, but this can often be just because they introduce a lot of overhead.

Rather than making your computation go faster, the systems introduce substantial overheads which can require large compute clusters just to bring under control.

In many cases, you'd be better off running the same computation on your laptop.

Methodology

Here is the set-up: we took several recent graph-processing publications from the systems community, and compared the measurements they report to simple single-threaded implementations running on my work laptop (RIP). We wrote competent implementations, but we didn't obsess deeply over fancy algorithms, cunning data-dependent tweaks, or what have you. I'll show you the code, and you decide.

We evaluated [PageRank](#) (20 iterations) and [graph connectivity](#) on two graphs, [twitter_rv](#) and [uk_2007_05](#). We chose these algorithms and datasets mostly because a recent OSDI 2014 paper [GraphX](#) used exactly these to evaluate several of the top systems, and we wanted to borrow their numbers rather than try to reproduce them on systems we aren't expert with.

As a caveat, these algorithms are quite specific to graph processing, and the data sets are not large (billions of edges, but still just a few gigabytes). Our conclusion is not that the systems we are going to look at are obviously bad, but rather that there is not yet much evidence that they are especially good.

The point we try to make in the HotOS submission is that evaluation of these systems, especially in the academic context, is lacking. Folks have gotten all wound-up about scalability, despite the fact that scalability is just a means to an end (performance, capacity). When we actually look at performance, the benefits the scalable systems bring start to look much more sketchy. We'd like that to change.

Measurements

All measurements other than the “single thread” lines are from the GraphX paper linked above. I’ll talk about the single-threaded implementations soon, but the results for the most direct implementations of PageRank and label propagation (the algorithm the systems use for graph connectivity) look like this:

Twenty pagerank iterations

System	cores	twitter_rv	uk_2007_05
Spark	128	857s	1759s
Giraph	128	596s	1235s
GraphLab	128	249s	833s
GraphX	128	419s	462s
Single thread	1	300s	651s

Label propagation to fixed-point (graph connectivity)

System	cores	twitter_rv	uk_2007_05
Spark	128	1784s	8000s+
Giraph	128	200s	8000s+
GraphLab	128	242s	714s
GraphX	128	251s	800s
Single thread	1	153s	417s

These single-threaded numbers paint the scalable systems in a pretty dismal light. With 128x as many cores, none of the scalable systems consistently out-perform a single thread at PageRank, which is probably one of the simplest graph computations (sparse matrix-vector multiplications). The systems are almost a factor of two slower than the single-threaded implementation for label propagation.

Unfortunately, it is only going to get worse for the scalable systems as our investigation continues.

Implementation

Let’s talk a bit about what the implementations look like, so that we can figure out whether we are being totally unfair. First, the measurements were taken on a laptop I no longer possess. I’ve reproduced most of the code in [Rust](#) on my new laptop, and the results are mostly the same (some faster, some slower). The measurements above are from a C# implementation, so this isn’t a “Rust is super fastestest” issue. The code is available on GitHub, including both [the C# version](#) and [the Rust version](#).

The algorithms we are talking about are all based on iterative scans of the edges in a graph. To support these algorithms, we are going to produce a few implementations of the following trait:

```
trait EdgeMapper {  
    fn map_edges<F: FnMut(u32, u32)->()>(&self, action: F) -> ();  
}
```

The `map_edges` method is generic with respect to a closure. It's a mouthful, but it means `action` can be called multiple times on inputs of type `(u32, u32)`, edges, and may modify its environment. This will probably make more sense once we see it used.

One nice thing about Rust is that each closure corresponds to a new type `F`, and a new specialized instance of `map_edges`, inlining the `action` method. In C# I had to manually unroll some loops. Ew.

Algorithms

Let's look at how PageRank gets implemented with respect to such a trait:

```
fn pagerank<G: EdgeMapper>(graph: &G, nodes: u32, alpha: f32) {  
    let mut src: Vec<f32> = (0..nodes).map(|_| 0f32).collect();  
    let mut dst: Vec<f32> = (0..nodes).map(|_| 0f32).collect();  
    let mut deg: Vec<f32> = (0..nodes).map(|_| 0f32).collect();  
    graph.map_edges(|x, _| { deg[x] += 1f32 });  
    for _iteration in (0..20) {  
        for node in (0..nodes) {  
            src[node] = alpha * dst[node] / deg[node];  
            dst[node] = 1f32 - alpha;  
        }  
        graph.map_edges(|x, y| { dst[y] += src[x]; });  
    }  
}
```

I'm not going to say very much about whether this is a good pagerank implementation. There are [better versions](#), but everyone uses this algorithm for benchmarking to ensure they are measuring their system's performance, not the algorithm's. This is textbook PageRank, no bells, no whistles.

Label propagation is basically just as simple. For those of you unfamiliar with the algorithm, each node maintains a label, and repeatedly improves it by asking their neighbors if they have a smaller label. This is run until convergence, at which point all nodes in the same component have the same label.

```
fn label_propagation<G: EdgeMapper>(graph: &G, nodes: u32) {
    let mut label: Vec<u32> = (0..nodes).collect();
    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]);
            }
        });
    }
}
```

These are pretty simple implementations. No tricks. Let's look at how we implement an `EdgeMapper`.

EdgeMapper implementation

Our first `EdgeMapper` implementation will be based on an adjacency list representation, which we will just map into memory and iterate over. There will be a smarter one later on.

We'll assume we have two files, `graph.nodes` and `graph.edges`, where

- `graph.nodes` contains a sequence of `(u32, u32)` pairs representing `(node_id, degree)`.
- `graph.edges` contains a sequence of `u32` values representing edge endpoints.

To enumerate the edges, we iterate through `graph.nodes`, where for each `(node_id, degree)` pair we read `degree` values from `graph.edges` and then pair each with `node_id` to form an edge.

Ignoring the `TypedMemoryMap` for the moment (it's 32 lines of wrapper around Rust's `MemoryMap`), the mapper structure and its implementation of `EdgeMapper` are:

```
pub struct NodesEdgesMapper {
    nodes: TypedMemoryMap<(u32, u32)>,
    edges: TypedMemoryMap<u32>,
}
```

```
impl EdgeMapper for NodesEdgesMapper {
    fn map_edges<F: FnMut(u32, u32)->()>(&self, mut action: F) -> () {
        let mut offset = 0;
        for &(node, count) in self.nodes.iter() {
```

```

        let limit = offset + count as usize;
        for &edge in self.edges[offset..limit].iter() {
            action(node, edge);
        }
        offset = limit;
    }
}
}

```

That's all! (except for the `TypedMemoryMap` type, available in the repository).

Getting smarter

Perhaps the single-threaded implementations above are a bit too simple. There are some very easy modifications that result in significantly improved performance. Remember how the simple single-threaded implementations were on-par with or maybe slightly better than the scalable systems? Here are the improved results using smarter algorithms (still from my old laptop and C#):

Twenty pagerank iterations

System	cores	twitter_rv	uk_2007_05
Single thread (simple)	1	300s	651s
Single thread (smarter)	1	110s	256s

Label propagation to fixed-point (graph connectivity)

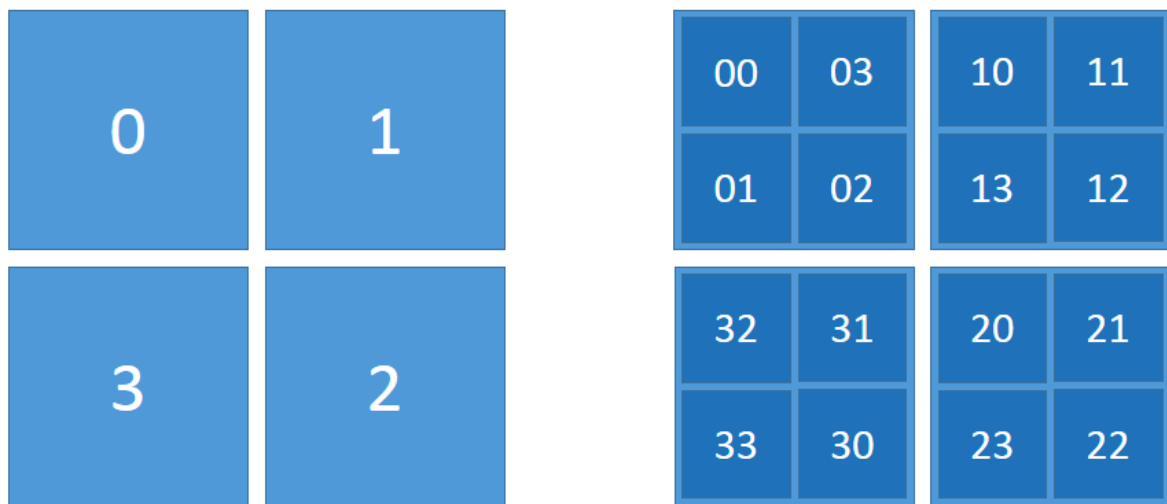
System	cores	twitter_rv	uk_2007_05
Single thread (simple)	1	153s	417s
Single thread (smarter)	1	15s	30s

There are two types of improvement, processing edges in different orders and using better algorithms, that put the single-threaded implementations so far ahead of the scalable systems.

Different data layout

The edge order used above processes all edges from one vertex before moving to the next vertex. Notice that neither of the algorithms need this to hold to be correct. A different ordering, one which exhibits much better cache locality, is the [Hilbert curve](#) ordering. Informally, it is like ordering edges (x,y) by the interleaving of the bits of x and y: you trade the great locality in the x coordinates to upgrade the poor locality in the y coordinate. You get (good, good) locality instead of (great, poor).

Visually, we can think of ordering pairs (x,y) as drawing a continuous line (a space-filling curve) through a 2d square. An initially square region is decomposed into four squares, each of which will be filled completely before moving to the next square. Notice that the sequence 0, 1, 2, 3 moves only to adjacent squares. The process is applied recursively within each square to order elements within, and even at the finer granularity the movements are only to adjacent squares, even when moving between squares at the coarser granularity. The ordering (curve) we produce is continuous, and very localized.



The Hilbert implementation of **EdgeMapper** is similar to our prior implementation, with some minor layout changes. We start from two files:

- **graph.upper** contains a sequence of `((u16, u16), u32)` values, each a 65536x65536 square indicating the upper 16 bits of x and y , followed by the number of edges in the square. These entries are ordered by the order their corresponding squares are visited by the curve.
- **graph.lower** contains a sequence of `(u16, u16)` values for the lower 16 bits for each edge. These entries are ordered by the the order the curve visits them when traversing their square.

The **EdgeMapper** implementation is basically identical to the implementation for vertex-order above, but results in improved PageRank running times. One lesson is that PageRank is essentially memory-bound; the time is spent either thrashing the TLB or waiting for results from the memory controller, rather than the CPU or disk. It scales because each machine offers an independent memory controller. Instead, you can partially fix the problem by re-arranging memory accesses like we have done here. You could also use [huge pages](#) (the C# code does), which I can't do yet on OS X.

We should also admit that the Hilbert curve ordering is not typically how graphs are presented, and consequently there would be some work involved in transforming your input data to this ordering. If you are only planning on doing one computation, it may not be worth

the effort. Fortunately, a single-threaded implementation takes less time to re-order the edges (about 180s in Rust) than any of the systems above take to perform 20 iterations of PageRank. It's included in the repository.

Both GraphLab and GraphX do their own data layout as part of pre-processing. One of the layouts they consider resembles the Hilbert ordering: partitioning vertices into k parts and sending sub-squares of edges to k^2 workers. The Hilbert-curve ordering is totally compatible with their system designs, and they could probably speed up a bit if they used it. It is somewhat in conflict with their approaches to sparse updates (when you only want to process a subset of vertices), trading latency in the sparse-update case for throughput in the (often more expensive) bulk-processing case.

Different algorithms

The label propagation algorithm is not commonly taught as a good algorithm for graph connectivity. It has poor asymptotic bounds, even when compared with more traditional [algorithms from 90 years ago](#).

Consider the standard [union-find algorithm](#), which maintains a `root` variable for each vertex, and scans the edges once, merging roots when it finds edges whose endpoints have different roots. We use the “weighted-union” variation, where the merge pays attention to how deep each root is when determining which way to swing the pointers (this gives it an $O(m \log n)$ asymptotic complexity).

```
fn union_find<G: EdgeMapper>(graph: &G, nodes: u32) {
    let mut root: Vec<u32> = (0..nodes).collect();
    let mut rank: Vec<u8> = (0..nodes).map(|_| 0u8).collect();
    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 },
            }
        }
    });
}
```

This is one more line of code than label propagation, but it is 10x faster and 100x less embarrassing.

The union-find algorithm is fundamentally incompatible with the graph computation approaches Giraph, GraphLab, and GraphX put forward (the so-called “think like a vertex”

model). That doesn't stop it from being way better than label propagation, and even [parallelizable](#).

Important things

Several factors affect the performance we've seen above. Let's see where the differences come from.

To get consistent measurements despite using `mmap`, each of these numbers are from a second run. This means that the data are effectively "in memory". The first runs are sketchier and more variable. We'll use graph connectivity as our example, from label propagation to our fastest union-find, running on my laptop (a 2014 MacBook pro) using Rust implementations.

150.6s : Label propagation and vertex ordering

31.1s : Switching to Union Find

16.0s : Switching to Hilbert-curve ordering

11.8s : Using unsafe indexing for roots

8.7s : Alternate EdgeMapper with smaller memory footprint (in the repo)

8.0s : Pre-fetching root[x] and root[y] before the while loop.

Note, these numbers differ from those reported above due to being a totally different implementation. Mainly, the first measurements took advantage of large pages, which do not seem to exist on OS X. This benefit is largely recouped by using the Hilbert ordering, which also eases pressure on the TLB. The custom EdgeMapper re-uses a small buffer, and likely messes up the TLB less than mmap does.

Take-aways

Lots of people struggle with the complexities of getting big data systems up and running, when they possibly shouldn't be using the systems in the first place. The data sets above are certainly not small (billions of edges), but still run just fine on a laptop. Much faster than the distributed systems, at least.

Here are two helpful guidelines (for largely disjoint populations):

1. If you are going to use a big data system for yourself, see if it is faster than your laptop.
2. If you are going to build a big data system for others, see that it is faster than my laptop.

We implemented a few of these algorithms in [Naiad](#) and were able to get performance that beats the single thread. Admittedly, it took some work for [parallel union-find](#), but we learned

quite a bit and made something better as a result (delightfully: we didn't have to change Naiad itself; just application logic).



We think everyone should have to do this, because it leads to better systems and better research.

Of course, there are some reasons why you might expect some overheads for your distributed system. The machines Amazon provides as VMs are differently spec'ed than my laptop, which targets peak single-threaded performance. GraphX uses hash tables (which can add an order of magnitude to the critical path) rather than dense arrays, to maintain its per-vertex state, to provide a layer of robustness. Fault-tolerance calls for data to be pushed from fast random access memory out to stable storage to insure against lost work in the case of a failure.

At the same time, it is worth understanding which of these features are boons, and which are the tail wagging the dog. We go to EC2 because it is too expensive to meet the hardware requirements of these systems locally and fault-tolerance is only important because we have involved so many machines, at which point we wonder whether the robustness of these hash tables is worth all the additional effort. Perhaps, but we should be thinking about it quite a bit harder than we currently are.

Frank McSherry

Frank McSherry
mcsberry@gmail.com

 [frankmcsherry](#)
 [frankmcsherry](#)

Some notes on doing less of what I used to do, and more of what I'd rather be doing.