

Bigger data; same laptop

Feb 4, 2015

This post follows up the previous post, [“Scalability! But at what COST?”](#), which got a great response. The short version of the previous post is that for the graph datasets and computations the scalable systems research community is currently looking at, a laptop outperforms the scalable systems.

There were several flavors of response to the post (many of which were very supportive!), but one that I’d like to address in this post is

One billion edges is just a few gigabytes; it’s hardly “big data” ...

My personal opinion is that if your system doesn’t perform well on one billion things, you don’t yet have the credibility to claim it’s going to work much better on a trillion things. But let’s see where this goes...

The only reported data I know of for graph processing at the “one trillion things” scale is [Facebook’s processing of a one trillion edge graph](#), where they got the per-iteration pagerank time down to under four minutes on 200 commodity machines (number of cores unknown to me). That’s really quite good (and, much better than Giraph performed in GraphX’s evaluation).

I don’t have a trillion edge graph, unfortunately, but thanks to the nice people at [Web Data Commons](#), I was able to download a [128 billion edge graph](#), based on the hyperlink structure from [Common Crawl](#). Using the same algorithms from the previous post yields a new (mostly empty) column:

Twenty pagerank iterations

System	cores	twitter_rv	uk_2007_05	common crawl
Spark	128	857s	1759s	unknown
Giraph	128	596s	1235s	unknown
GraphLab	128	249s	833s	unknown
GraphX	128	419s	462s	unknown
Single thread	1	242s	256s	46600s*

*: An earlier version had 23,653 seconds as the number for pagerank. Unfortunately, the code was set to do only ten iterations (in order to compare to GraphX, whose scaling measurements are for only ten iterations). We apologize for the error.

Graph connectivity

System	cores	twitter_rv	uk_2007_05	common crawl
Spark	128	1784s	8000s+	unknown
Giraph	128	200s	8000s+	unknown
GraphLab	128	242s	714s	unknown
GraphX	128	251s	800s	unknown
Single thread	1	15s	30s	1700s

Importantly, these algorithms are *exactly* the same as in the previous post. Nothing has been changed, other than (a few sections down) a different on-disk representation of the graph so that it fit on my SSD. The edges per second are mostly within a factor of two across the graphs; this approach even scales!

Note: The single-thread numbers for pagerank on the smaller graphs are not the same as in the previous post. Because the common crawl dataset is so large, I was only able to process it in Hilbert curve order. It seemed appropriate to use the corresponding numbers for the smaller graphs, rather than the vertex order numbers.

A COST challenge for scalable systems

I'd like to challenge the teams behind the existing graph processing platforms to evaluate and report their COST (Configuration that Outperforms a Single Thread) for this dataset. The dataset is nearly two orders of magnitude larger than those most currently use for system evaluation, and yet still nearly one order of magnitude *smaller* than can be easily processed by my laptop. It is much “bigger” data than we've been using previously, and it is available for everyone to download, so get to it!

The only partial data we have is for Giraph, for which 200 machines is more than enough. I don't think we know very much about any other scalable graph processing system. Let's fix that!

I'll happily post updates for performance numbers on specific systems. To make it fun, let's say that each submission gets to name another system whose row is “unknown” until its measurements arrive.

Implementation notes

There were some relatively minor implementation details involved in getting 128 billion edges onto my laptop, and then processed efficiently (which isn't to say that I got them right the first time). Doing the math, if one writes down a `(u32, u32)` for each edge, this ends up at a terabyte of data, and my laptop doesn't have that kind of SSD. So, we'll have to be a bit smarter.

Using the Hilbert curve layout leads to a very simple compressed representation: having sorted the `u64` values it uses to represent edges, one can simply delta-encode them (writing the differences between adjacent values). Because the Hilbert curve teases out locality, the gaps are often quite small, and therefore compress nicely. When I combined a variable-length integer encoding with gzip, the resulting file was only 45GB; about 2.8 bits per edge). Decompressed (with just the variable-length delta encoding), it is 154GB, which is the representation I used to get the measurements above.

Not only is the Hilbert curve good for compression, it is pretty much mandatory due to the locality it provides when accessing the per-vertex state. Random access into main memory is not all that bad, but when the working set for vertex state in pagerank is 40GB, random access turns in to thrashing. This is not only slow, but it wears out my SSD, and no one wants that. The near-linear performance scaling as we went up two orders of magnitude in size is evidence for how cool the Hilbert curve is.

Transforming the data to the compressed Hilbert curve encoding was relatively efficient. My laptop was able to transform (encode, sort, and compress) the data faster than I could download it, so I didn't work too hard to optimize the performance. However, this is an excellent example of where parallelism can help: each of the [700 files comprising the graph](#) could be processed independently (and I did fire up a second process on occasion). If you had 100 cores (like most of the scalable systems use) the whole conversion process would just take a few minutes.

The implementation is available in the [COST repository](#), though the documentation is not yet brilliant. If you have questions, especially if you'd like to be doing this sort of thing in your scalable system and want pointers, just shoot me an email.



What's next

I'm not going to try this nonsense on any larger graphs. The ball is now in the court of the scalable systems. We'll just have to wait and see how they do before setting any stretch performance goals.

Instead, it's time to get back to work on [timely dataflow in Rust!](#) Stay tuned for that kind of post.

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.

