Tradeoffs Between Synchronous and Asynchronous Engines in PowerGraph

Joshua Send (js2173)
Trinity Hall

February 22, 2018

Abstract

Distributed graph processing system exist with both synchronous or asychronous execution modes. While more recent graph processing systems largely to favor the synchronous model, some works attempt to utilize asynchronous execution as well. However, there is a lack of thorough evaluation of the costs and benefits of each model. While the intuition might be that the asynchronous model allows problems to be solved faster, at the cost of determinism and understandibility, the reality is much more muddled and complex. This project aims to examine further asynchronous execution in comparison to the synchronous model, and whether combined synchronous and asynchronous schemes can be useful.

1 Introduction

This project utilizes PowerGraph [4], an open source distributed graph processing system [1]. It introduced innovations such as vertex-cuts, and a highly parallelizable, general Gather-Apply-Scatter paradigm for defining computation on vertices. Further, it supports two execution modes for graph computation, synchronous and asynchronous, the focus of this work.

Many contemporary, widely used graph processing frameworks,such as GraphX [7] and Apache Giraph [2], utilize the synchronous execution model, and do not provide asynchronous engines. One interesting development in 2014 was PowerSwitch [6], which combines both models within a hybrid PowerGraph engine. It uses online sampling or neural network based prediction to estimate whether the other execution mode is faster, and switches modes if necessary. This combined engine, and whether it can be profitably used, is evaluated along with the standard modes.

The upcoming evaluation focuses on effectiveness of execution modes as cluster size varies (between 2 and 40 commodity machines), and as the proportion of resources dedicated to graph processing versus networking is varied (from 50% to 100% graph processing). Both of these aspects are under-evaluated in both the PowerSwitch and PowerGraph papers.

[1]Largely because later work builds a hybrid synchronous/asynchronous executoin mode on top of PowerGraph

2 Difficulties

One peculiar difficulty with this project is that GraphLab (of which the 2.1 version is PowerGraph) was spun off into a company, re-branded, and then was finally bought by Apple. Their original website no longer exists, which hosted dependencies, help forums, and documentation. All that remains is a GitHub repository [2] with outdated and buggy source code, and has only scattered documentation. Significant amounts of time were invested into finding the correct dependencies and fixing bugs. The task of reconstructing a compatible Amazon EC2 Linux image was also rather tedious.

3 Overview of Execution Modes

3.1 Synchronous

The synchronous execution model uses a global compute barrier to allow an *active set* of vertices to finish computation totally, before letting the next active set begin computing. This is very similar to the classic BSP model [5], where each iteration is called a superstep.

One useful feature of the synchronous mode is that network resources can be utilized more effectively, by batching messages leaving vertices together. This means that the synchronous mode is likely to perform very well for I/O heavy computations, such as PageRank, where the actual processing time is small for each vertex. However, the global barrier means that stragglers have a large impact on the efficiency of the system. To amortize this cost, having a large set of actively computing vertices should be advantageous, while asymmetric computation loads, where few nodes do work at a time means the cost is not well amortized.

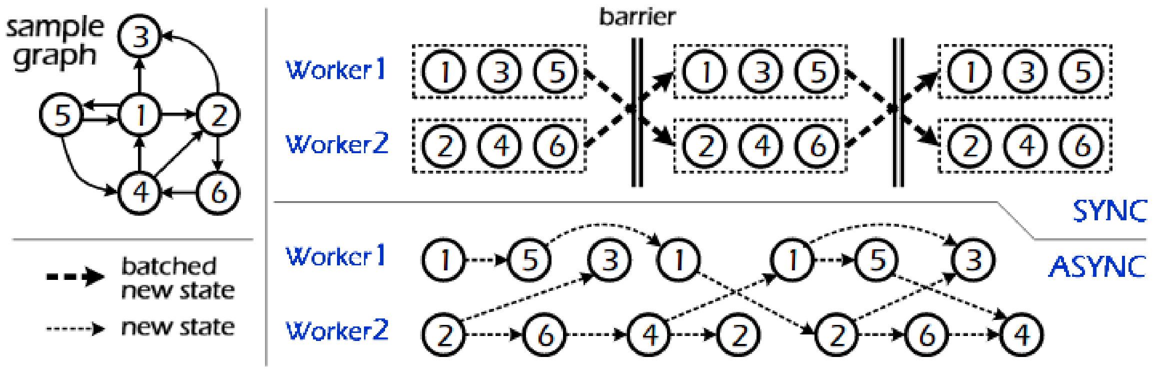


Figure 1: Diagrammatic comparison between synchronous and asynchronous execution.

4 Asynchronous

The second engine defined in PowerGraph does away with the global compute barriers and supersteps to allow vertices to begin calculations as soon new data is available at some

[2] https://github.com/xiechenny/powerswitch

neighbor. While it seems like this would automatically unlock higher throughput and solve problems faster, this is not the case: while higher throughput can be achieved, much of it is recalculating values based on partially stale values. Thus, the final result may not be converged to as quickly. Two further difficulties are that not all algorithms can be rewritten to operate correctly in this nondeterministic mode, and when correct algorithms are possible, they are more difficult to reason about.

Note that a few problems, such as graph coloring, might only converge with some nondeterminism, and may alternate between two possible colorings in the synchronous mode. A further contrast is that I/O can no longer be batched and used efficiently – we can hypothesize that problems that are computation heavy, rather than communication heavy, will be suited to asynchronous execution.

Another downside of the asynchronous mode is specific to vertex-splitting systems such as PowerGraph: due to replicated vertices, locking is required when updating data. Locking would also be required if data on vertices enters invalid states while being updated - reading data from a neighbor that is also active would mean some kind of lock. Hence, if there are many active nodes or a very highly connected graph, there could be a lot of resource contention and waiting to obtain locks.

5 Hybrid

PowerSwitch [6] innovates on PowerGraph by combining both the synchronous and asynchronous execution modes, switching between modes to take advantage of their relative strengths. From the previous sections we see that different problems and system configurations might work better in one of the modes. Even within one problem, the preferred execution type may vary: when executing single-source shortest path, the asynchronous mode performs best at the beginning and end when the active sets of the problem are small, and synchronous has better performance in the middle of the problem [1].

The authors of PowerSwitch claim that the only two relevant measurements needed to estimate the efficiency of a mode is the throughput (ie. vertices processed per second), and the convergence ratio μ (how efficient the computation is in asynchronous versus synchronous mode). The authors set μ to around 0.75 for PageRank and 0.5 for SSSP, which seem like values pulled out of thin air. However, otherwise this method appears sound.

The task of deciding when to switch modes thus boils down to estimating the throughput of the next iteration (synchronous) or time delta (asynchronous), for both the current mode and the alternative one. The current mode is handled with a running average. The alternative mode is predicted by either running a subset of the problem at system initialization and measuring throughput, or using a neural network predictor (not tested here). The cost of doing the profiling on a subset of the input counts is part of the switching overhead. What remains to be seen is whether the gain from switching modes outweighs the profiling and switching costs.

6 Algorithms

The evaluation focuses on PageRank and single-source shortest path as the core problems. I also test a non-deterministic problem, alternating least squares which was not evaluated in

[6]. More would have been explored given more time and Amazon EC2 credit (for instance, a very CPU heavy computation such as loopy backpropagation).

6.1 PageRank

PageRank is a classic distributed algorithm which lends itself well to both synchronous and asynchronous execution. The PowerGraph algorithm for PageRank is given below in Listing 1. This illustrates the Gather, Apply, Scatter paradigm: the gather step is parallelizable function which for each neighbor collects the current normalized rank. The Apply step rewrites the vertex's PageRank, and then the neighboring vertices are notified if the value has converged sufficiently.

Listing 1: Pseudocode implementation of PageRank in PowerGraph

```
/* Select which edges to utilize */
gather_edges(vertex) {
  return graphlab::IN_EDGES;
}

/* Gather the weighted rank of the adjacent page   */
gather(vertex, edge) {
  return (edge.source().data() / edge.source().num_out_edges());
}

/* Use the total rank of adjacent pages to update this page */
apply(context, vertex, total) {
  const float newval = (1.0 - RESET_PROB) * total + RESET_PROB;
  vertex.data() = newval;
  if (ITERATIONS) context.signal(vertex);
}

/* The scatter edges depend on whether the pagerank has converged */
scatter_edges(context, vertex) {
  if (ITERATIONS) return graphlab::NO_EDGES;
  if(USE_DELTA_CACHE || std::fabs(last_change) > TOLERANCE ) {
    return graphlab::OUT_EDGES;
  } else {
    return graphlab::NO_EDGES;
  }
}

/* The scatter function just signal adjacent pages */
scatter(context, vertex, edge) {
  if(USE_DELTA_CACHE) {
    context.post_delta(edge.target(), last_change);
  }
  if(last_change > TOLERANCE || last_change < -TOLERANCE) {
      context.signal(edge.target());
  } else {
    context.signal(edge.target());
  }
}
```

6.2 Single-Source Shortest Path

The distributed, parallel algorithm for SSSP is roughly known as the distributed Bellman-Ford algorithm. This algorithm is interesting since the active vertex set begins very small, then increases to a maximum, and then decreases back down to 0. As Xie et al. mention, SSSP is precisely the kind of problem for which the hybrid engine excels, taking advantage of small active sets using the asynchronous mode, and using synchronous execution in the middle of the problem when there are many active vertices.

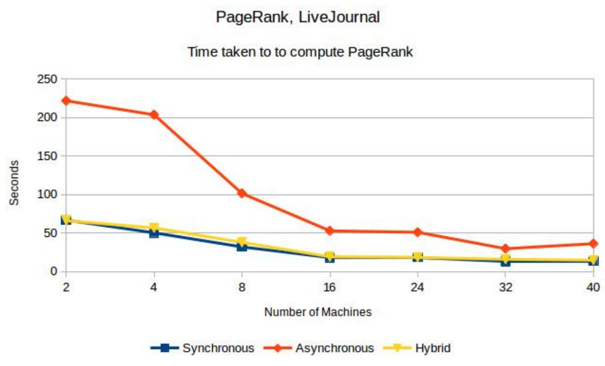
6.3 ALS

Alternating least squares is a type of collaborative filtering, and more specifically a type of non-negative matrix factorization. The problem is, given a data matrix X, we wish to decompose it into two matrices W and H such that $X \approx WH$. The difficulty is that the matrix X, which might represent users on one axis, and song selection frequency on the other, is incomplete. The goal is to minimize the reconstruction error $\sum_{ij}(X - WH)^2_{ij}$. This problem is NP-hard, and the ALS approach to solving it is iterative in nature, and may not always converge [3]. I include it as a brief empirical examination of the effect of execution mode on convergence.

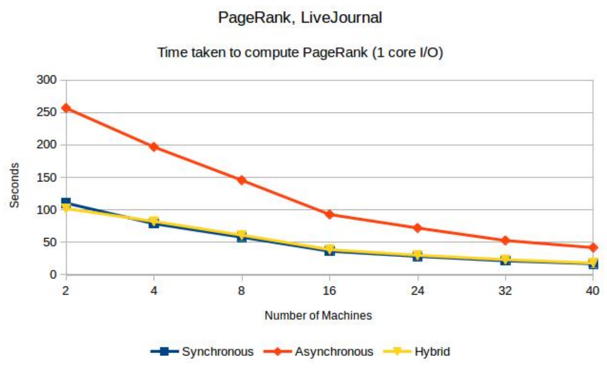
7 Evaluation

The data presented in the following subsections is obtained from scaling Amazon M4.Large instances from between 2 and 40 machines. The M4.Large configuration contains 2 CPU cores, 8 GB of RAM, and 450 MB/s network throughput. The backing storage was of the SSD type. To allow a larger variety of experiments to be run, the only 2 repetitions of each configuration were executed; the mean of these repetitions is presented in the data. Graphs were partitioned using the 'Grid' method where possible, and otherwise the 'Oblivious' method. Both offer lower vertex replication than a random distribution.

7.1 PageRank



(a) Using 2 cores for computation.



(b) 1 core for computation, 1 core for networking.

Figure 2: PageRank on LiveJournal Dataset

The plots in Figure [**fig:pr**] shows the performance of the synchronous, asynchronous, and hybrid modes computing the PageRank of the LiveJournal dataset [3]. The asynchronous mode is significantly slower than the others, taking over 200 seconds to compute the PageRank. On the other hand, the synchronous and hybrid modes are neck and neck. Hybrid mode execution logs show that while the asynchronous mode is entered at one point, the vast majority of the time is spent in synchronous mode. In most cases, the hybrid mode performs slightly worse than pure synchronous mode, though this seems to be a constant difference and can be attributed to the sampling, prediction, and switching overhead.

An interesting result is observed when plotting the relative slowdowns when dedicating 1 core to I/O, and the other to the graph computation, versus 2 cores for graph computation and none to networking (Figure **??**). Synchronous mode generally takes a larger performance hit when dedicating a core to I/O. This means that the synchronous mode is requires less power for communication (due to batching), and making better use of the cores to do graph computation. This effect gets diluted across more machines, as each processor is doing less work and more I/O.

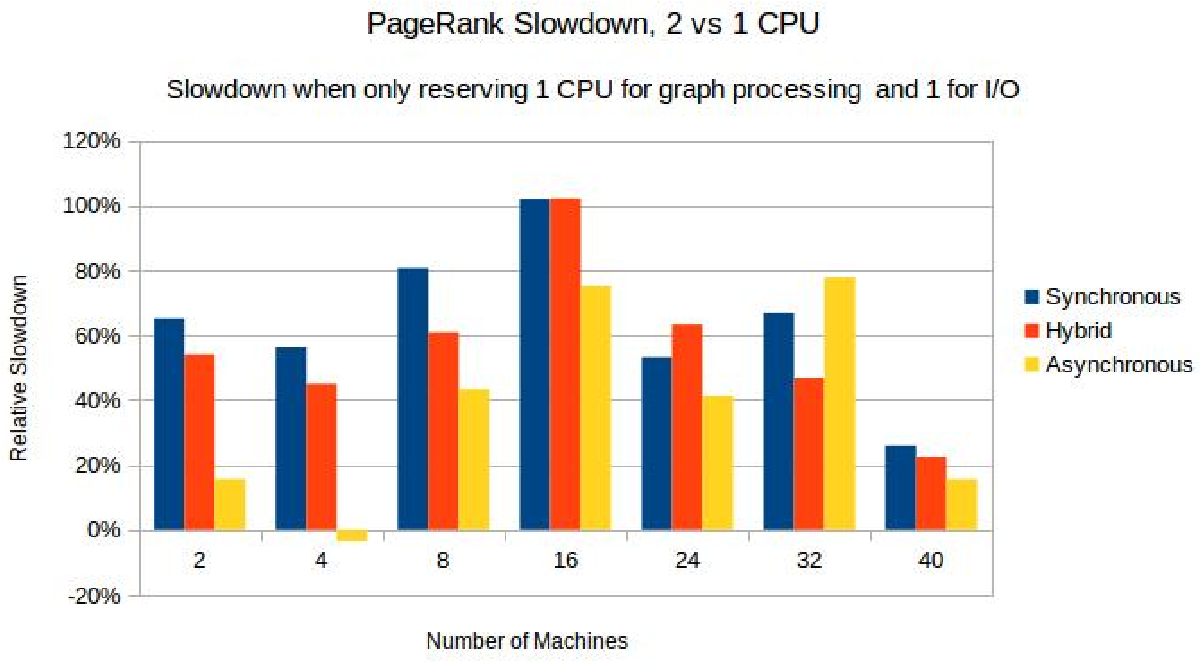


Figure 3: Relative slowdown from shifting 1 CPU core to networking rather than graph processing.

One last thing to observe is the throughput of the two modes, shown in Figure **??**. Throughput of synchronous mode is consistently higher than asynchronous: contrary to the initial hypothesis that asynchronous mode generally has higher throughput. In this case, it can be attributed to the locking required due to vertex cuts.

[3]`https://snap.stanford.edu/data/soc-LiveJournal1.html`

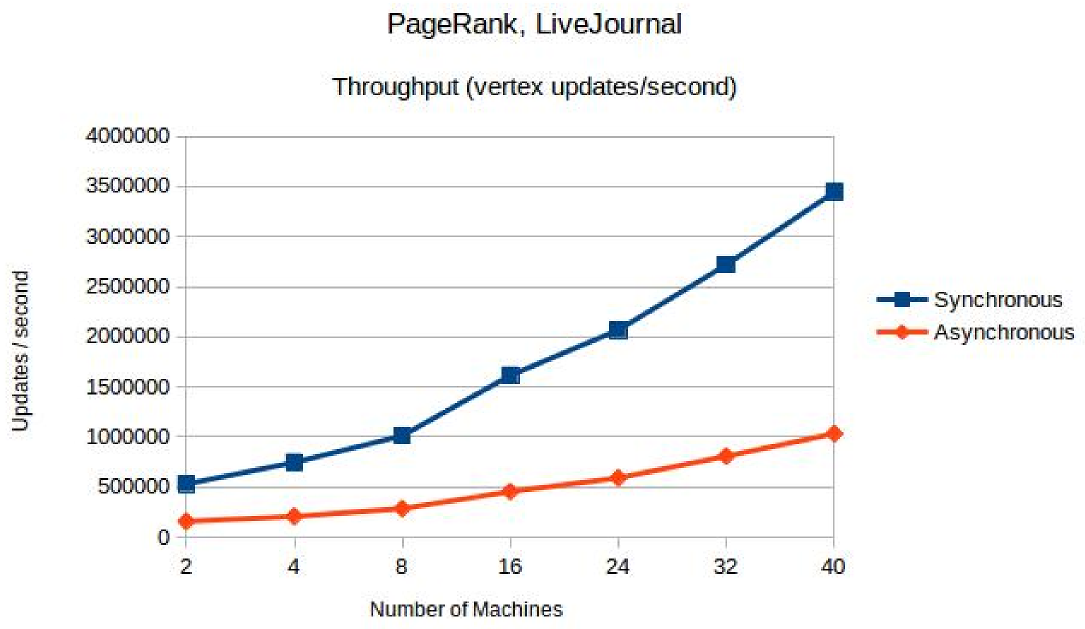
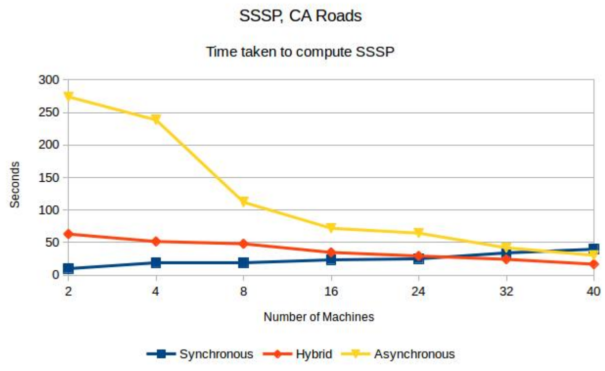
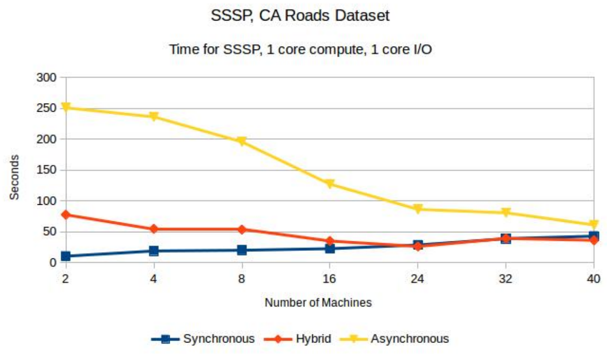


Figure 4: Throughput of asynchronous and synchronous mode for the PageRank dataset. Shown is throughput using 2 cores for graph computation.

7.2 SSSP



(a) Using 2 cores for computation.



(b) 1 core for computation, 1 core for networking.

Figure 5: SSSP on CA Roads dataset.

Xie et al. in the PowerSwitch paper demonstrate that with 12 machines, the SSSP problem on the CA Road dataset [4] is significantly faster in Hybrid mode, switching from asynchronous mode after the initial active vertex set grows large enough, then switching back into asynchronous mode after the active set shrinks again. While this switching behavior is observable from the logs, no significant speedup is visible until 24 machines are used in the 2-core case, and 40 machines in the 1-core case, as shown in Figure 5. It is also interesting to note that any larger number of machines than 2 slows down the SSSP problem

[4]`https://snap.stanford.edu/data/roadNet-CA.html`

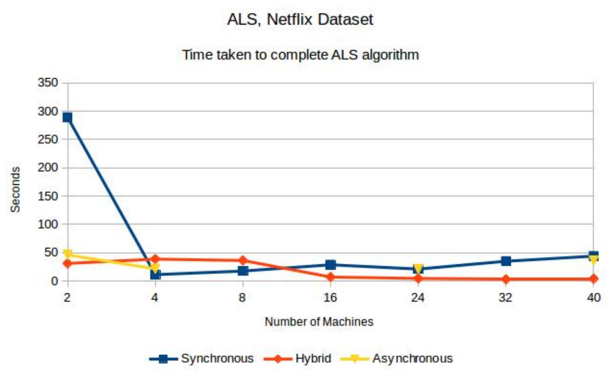
in synchronous mode. This indicates a major flaw in PowerSwitch and a possible future work direction, which is that over-provisioning resources should not affect performance.

Overall we deduce, that the SSSP problem is not conclusively faster in hybrid mode until scaled to a large number of machines, and even then can easily be beaten by a small number of machines in synchronous mode, a fact not described by previous work.

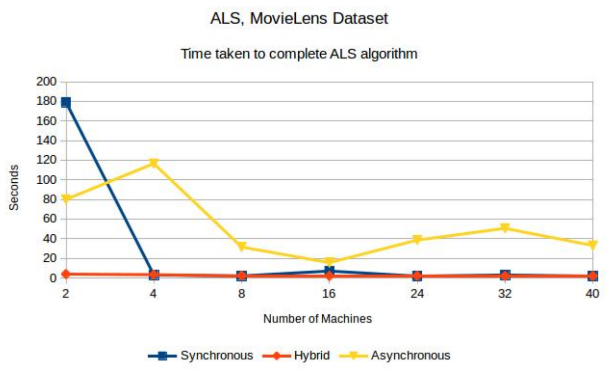
Several differences may explain the discrepancy between previous results and those presented here: the M4.Large instances from Amazon EC2 represent relatively standard, commodity machines. We have seen that the asynchronous mode can benefit from more resources dedicated to networking – it may be the case that the authors of PowerSwitch use higher speed network links, machines with more computational power dedicated to networking, or a different version of the MPI library which executes the distributed algorithms. However, the data presented here shows that on commodity hardware, asynchronous mode using OpenMPI is no faster than synchronous mode.

7.3 ALS

Figures 6 show the time taken for the ALS algorithm to complete on the Netflix (vertices: 68958, edges: 499405) and MovieLens (vertices: 9746, edges: 1000209) from Stanford SNAP. The most interesting result from these plots are that the Hybrid mode actually performs better than the others with only 2 machines; logs show the algorithm begins for a short time in asynchronous mode before switching to synchronous. This seems to induce faster convergence than in purely synchronous mode. Further study into the details of ALS would be required to fully explain this.



(a) ALS for Netflix Dataset



(b) ALS for MovieLens Dataset

Figure 6: Runtime of ALS algorithm on different datasets, using 2 CPU cores for computation.

A second thing to note is the lack of data for 8, 16, and 32 machines in asynchronous mode and Netflix data: at these points the ALS algorithm actually diverged to an a error of $9.99e+99$. Across all ALS Netflix experiments, the asynchronous mode diverged a total of 8 out of 28 times; in hybrid and synchronous mode the error always converged to around 0.30343. For the MovieLens dataset, which has far fewer vertices and more edges, asynchronous mode only diverged 1/28 times, but the other modes never diverged. For this algorithm, we conclude asynchronous mode offers neither speed nor convergence advantages. When converged, all errors were within ± 0.1 of the value given above.

8 Conclusion

As evidenced in Section 7.1, asynchronous mode actually has lower throughput when computing PageRank, contrary to the conventional wisdom that asynchrony unlocks higher throughput. It was also found that asynchronous mode benefits if more computational power is dedicated to networking – reflecting the comparatively worse I/O efficiency that was hypothesized.

The hybrid engine was found to rarely offer performance advantages. It was almost never significantly faster than using solely synchronous mode. Notable is the lack of speedup from using the hybrid engine for SSSP (Section 7.2), contradicting the 1.45X - 2.29X speedup found in [6]. This stems from the under-performance of the asynchronous mode in SSSP, which may be attributed to using commodity hardware rather than high performance clusters.

This work concludes that where possible, synchronous mode offers more consistent and better performance. Asynchronous mode's touted higher throughput may be being negated by the locks required to keep data consistent and distributed data structures such as a global scheduling queue. The hybrid engine switches modes as expected, eg. from asynchronous to synchronous mode and back when the active sets in SSSP grow and shrink, but this fails to yield expected performance gains.

Further study attempt to use higher performance hardware configurations; the parameter space here is quite high: varying number of processors reserved for computation versus I/O, and faster disks and networks could all be explored. Alternatively, the characteristics of problems could be explored further: synthetic datasets which explore input graph characteristics could be created – for instance, the density of the graph could have a large impact on locking and network usage in asynchronous mode.

References

[1] Dimitri P Bertsekas, Francesca Guerriero, and Roberto Musmanno. "Parallel asynchronous label-correcting methods for shortest paths". In: *Journal of Optimization Theory and Applications* 88.2 (1996), pp. 297–320.

[2] Avery Ching. "Giraph: Production-grade graph processing infrastructure for trillion edge graphs". In: *ATPESC, ser. ATPESC* 14 (2014).

[3] Nicolas Gillis. "The why and how of nonnegative matrix factorization". In: *Regularization, Optimization, Kernels, and Support Vector Machines* 12.257 (2014).

[4] Joseph E Gonzalez et al. "PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs." In: *OSDI*. Vol. 12. 1. 2012, p. 2.

[5] Leslie G Valiant. "A bridging model for parallel computation". In: *Communications of the ACM* 33.8 (1990), pp. 103–111.

[6] Chenning Xie et al. "Sync or async: Time to fuse for distributed graph-parallel computation". In: *ACM SIGPLAN Notices* 50.8 (2015), pp. 194–204.

[7] Reynold S Xin et al. "Graphx: A resilient distributed graph system on spark". In: *First International Workshop on Graph Data Management Experiences and Systems*. ACM. 2013, p. 2.