Graph Partitioning for Distributed Shared Memory

Clement Fung
cfung1@cs.ubc.ca

Stewart Grant
sgrant09@cs.ubc.ca

Abstract

The modern internet generates petabytes of data per day. Processing vast amounts of data is an increasingly common task both for scientists and modestly experienced programmers. Often this data is naturally represented as a graph, such as social media networks, webpage links or city networks, and requires clusters of machines to process. Concurrent trends in data centre architecture suggest that the rack is the new server, and shared memory is now a feasible interface between collocated rack servers. These trends made us wonder: *How simple can fast graph processing be on a rack of servers?*. We investigated the tradeoffs of the conventional Pregel *"think like a vertex"* programming model, and found its performance unacceptable. In contrast, we explored the merits of a *"Think like a subgraph"* model, which respects graph locality in common graphs, provides a more holistic programming interface, and runs faster!

1 Introduction

Big data processing is complicated. Scientists and experienced programmers alike struggle with managing and configuring clusters of machines to processes large amounts of data. Frameworks like Hadoop and Pregel have significantly eased the difficulty of big data processing, but they remain intimidating for the layman. We noticed a trend in scientists and researchers, finding that they wanted to *"Just write python code that ran on a bunch of computers"*. For the benefit of science we investigated how to make this dream a reality.

Running all Python code on clusters of machines is impractical would lead to sluggish un-optimal code, due to immense network overhead. Instead, we concentrated our efforts on a common but difficult big data processing task: graph processing. Many frameworks exist for distributed graph processing [19, 6, 15, 18, 25, 9], and many general frameworks exist which are used for graph processing [24, 26, 13, 21]. These frameworks vary in their complexity, but none are *"accessible"* for programmers that lack relevant experience.

With the exception of [15], the aforementioned systems suffer a common pitfall to accessibility; they expose the complexity of a distributed message passing system to the user. Extensive work has been done to hide this complexity in the abstraction of distributed shared memory (DSM) [14, 22, 20, 10, 12]. The benefits of DSM have been ignored in recent years due to its flaws, most notably fate sharing and sub optimal performance. Dismissing DSM may have been a shortsighted mistake. Ultra dense memory and the approach of terabit-level bandwidth within a rack give modern racks the appearance of a single machine, and has lead towards disaggregated architectures [1, 2, 3, 5, 11]. Such futuristic systems lend themselves naturally to DSM which motivates our proposal for a corresponding computation framework.

The largest disadvantage of DSM is performance. Programmers can write terribly performant programs by failing to reason about the location of memory, leading to memory thrashing. In computation where a high degree of consistency between shared resources is needed, DSM is the wrong tool for the job. In contrast, when large amounts of computation can be performed between memory synchronizations, DSM provides a simple and efficient programming model. Graph processing suffers from a lack of locality. In computations such as PageRank, a single iteration may perform edge updates which require the synchronization of all machines in a cluster. This problem can be largely avoided in practice by carefully pre-processing graphs into equally-sized partitions, where the minimum number of graph edges exist across machines. The cost of pre-processing a graph can be large; in some cases, the complexity of finding a good partition is greater than solving the initial problem! Here we demonstrate that the cost of graph partitioning is worth it for the benefits that DSM provides.

In this paper we attack the problem of developing a simple and efficient graph processing interface for DSM. Specifically we make the following contributions.

- A simple graph processing API for DSM
- A graph partitioning scheme optimal for DSM

- An evaluation of processing performance between partitioned DSM processing, and Pregel-style graph processing

The remainder of this paper is organized as follows. In Section 2 we discuss approaches and strategies for scalability. In Section 3 we methodology for automatically scaling up graph processing. Section 4 describes other systems similar to ours, Section 5 discusses future work, and Section 6 concludes the paper.

2 Scalability

3 Auto-Scaling

3.1 Why Distributed Shared Memory?

Given the results from the scalability analysis, we aim to build a system that transparently scales for practitioners that desire performance, yet want minimal changes to their single-threaded interface.

What options exist for this setting? Implementing a full MPI-style system would allow for the highest potential gains in performance, but is not generalizable and would be intrusive to the programmer's interface. One could also opt for a system such as Naiad [21] or Hadoop [24]. While simplier to reason about, this also requires significant setup and is still requires a large change in the programming interface.

Instead, we turn to the next-simplest abstraction: distributed shared memory (DSM). DSM comes with the advantage that its programming model is similar to the single machine through a key value interface, [22], and hides the complexities of the network from the developer. DSM seems like an ideal solution for the problem presented, but has traditionally been shown to lack the required performance, and gives the developer little control over locality.

Thus, we propose BlueBridge, a locality aware DSM system. BlueBridge includes an API for access and placement of distributed shared memory, which aims to alleviate the concerns previously posed by DSM. With current trends in rack scale datacenters, disaggregated datacenters are evolving into dense memory systems with terabit-level internal bandwidth. These environments support the conceptual idea of DSM, in which a rack scale system poses as a single machine.

3.2 BlueBridge Design

BlueBridge is designed for a simple interface, that hides the complexity of shared memory from the developer. The intended users of BlueBridge are those who have a need to process large datasets, but do not want to reason about the complexities of distributed shared memory. We choose to provide an interface to BlueBridge through Python, using a shared dictionary interface. This API provides ability to perform *put(key, value)* and *get(value)* on a Python dictionary, while having all the required consistency and messaging capabilities hidden.

We simulate a DSM system by using Python 2.7, and its built-in *Manager* library. The *Manager* library allows for simple management of shared state between processes by providing interfaces to shared arrays, dictionaries and easy locking semantics.

Continuing the theme from the COST investigation, we choose to target graph processing as an example use case for BlueBridge. We also use iGraph 0.6, which includes a flexible interface for graph processing in Python. iGraph performs its computation in C and is interfaced with Python, so it came to be a natural choice given the design constraints.

3.3 Experimental Setup

In order to evaluate the parallel graph processing of BlueBridge, we computed PageRank on three graphs, collected from the Stanford Large Network Dataset Collection. [17] These graphs will be referred to as:

- wikiVote: A Wikipedia voter network, with 7115 vertices and 103689 edges

- GrQc: A citation network of General Relativity/Quantum Cosmology papers, with 5242 vertices and 124496 edges

- condMat: A citation network of Condensed Matter papers, with 23133 nodes and 93497 edges

What is most important of these three graphs is that they have varying degrees of connectivity. GrQc can be split into 2,4 or 8 partitions with a relatively low degree of cutting, while wikiVote is much more difficult to cut.

Graph	2 part.	4 part.	8 part.
GrQc	3.38%	4.93%	7.20%
condMat	6.56%	13.38%	18.69%
wikiVote	25.45%	50.05%	64.79%

Table 1: Percentage of edges cut when splitting into 2,4 or 8 partitions.

3.4 Partitioning

Graph partitioning occurs as a pre-processing step when performing graph processing on DSM. In order to respect locality within DSM, our partitioning algorithm aims to provide an optimal environment for the parallel processing of graphs. We define an optimal partitioning scheme as one in which the number of vertices are roughly equal for each partition, while the number of edges that cross partitions is minimized. We leverage the METIS [16] library, an MPI implementation for parallel partitioning of large graphs, to determine an optimal partitioning for the system. METIS simply accepts an adjacency list, which can be built using iGraph library funcitons, and outputs the the graph partitioning scheme as a membership vector. METIS is built in and runs in C, and the corresponding Python binding, PyMetis [4] was used to determine the optimal graph partitioning once it is read into memory. The time to partition the graph is several orders of magnitude less than the total processing time.

Graph	Time To Partition (avg.)
GrQc	0.027s
condMat	0.151s
wikiVote	0.050s

Table 2: Time taken to partition graph with PyMetis

Naturally, partitioning a graph will cause edges in the graph to cross partitions. Applying techniques seen in [7] and [8], we use a ”mirroring” technique for storing edges between partitions. For each edge that crosses partitions, only vertices on two separate partitions will be affected. The partition which contains the destination vertex will be considered the master of the given edge. All edges which connect to the master vertex from a different partition will create a ”mirror” vertex. These edges are stored in shared memory, such that they can be written to and read from different machines. Master vertices maintain references to the shared edges on which they depend. The partitioning procedure is shown in Figure 1.

For the 3 graphs described above, 20 iterations of PageRank were performed on BlueBridge, both with the partitioning scheme described above, and a random assignment of nodes. Pregel uses a hash of the vertex ID for node assignment, [19] which is not locality aware and creates unnecessary external edges. In a DSM environment, leveraging locality can lead to significant speedups. GrQc, which was the most easily partitioned graph, demonstrated a 10x speedup, while the wikiVote graph only showed speedups between 1-2x. These results are shown in Figure 2.

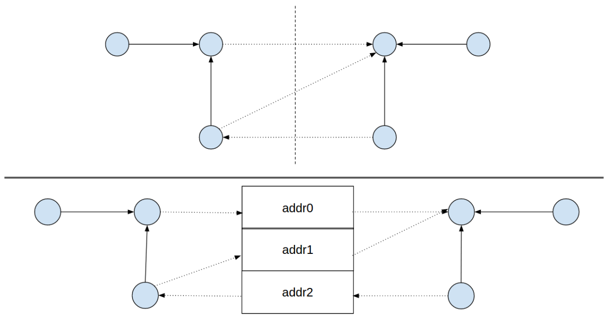

Figure 1: An example of the partitioning procedure. Shared edges are loaded into shared memory.

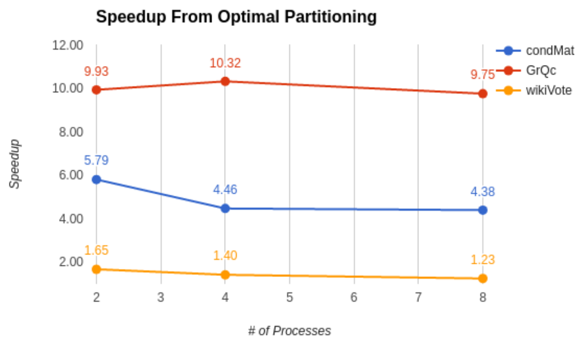

Figure 2: Speedups attained by using optimal partitioning. Speedups are relative to the performance of their corresponding random partitioning scheme.

3.5 RDDs over Supersteps

Typical graph processing algorithms rely on a ”superstep” mechanic, where for each iteration, all vertices are synchronized together before proceeding to the next iteration. When only a small subset of the edges in a partitioned graph are placed in distributed shared memory, creating a persistent record of edge weights on previous iterations becomes more feasible. This was explored by allocating additional addresses in the simulated shared memory for previous iterations, and storing them like resilient distributed datasets (RDDs). Access to this shared memory was performed in a publish and subscibe pattern of access. Thus, when performing a pagerank iteration, blocking only occurred on a read operation if the edge weight was not yet updated for the desired iteration.

We tackle the performance concerns of DSM by implementing this mechanism in BlueBridge. Speedups of approximately 1.5 - 2.1x wer observed, for both smart and random partitioning, varying degrees of graph partitioning, and different numbers of processes. The results of this experiment can be seen in Figure 3.

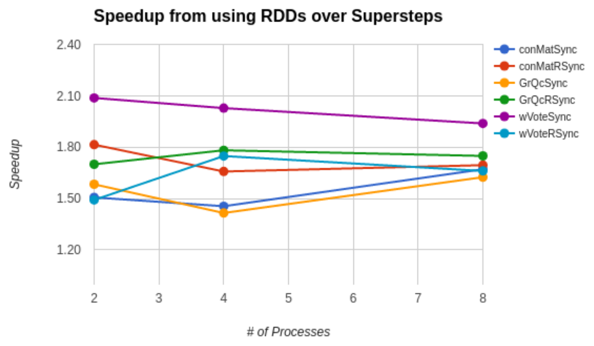

Figure 3: Speedups attained by using an RDD style of storage, instead of traditional supersteps.

Also not explored in this setting, the use of RDDs for shared memory implies the potential for resilience in the face of machine failures. Failed instances of BlueBridge could potentially restart and pull its dependent values from the globally shared memory, preventing redundant computation.

4 Related work

The concept of using shared memory in parallel processing of graphs has been explored by Shun, who created Ligra: a graph processing framework for shared memory. [23] Ligra uses a vertex-subset level of consistency, and therefore fails to reason about the overall structure of the graph. One could use Shun's system in a fashion similar to BlueBridge by mapping the partitioning scheme onto the VertexMap design of Ligra. As Ligra processes vertices, it uses a frontier-based approach, performing vertex-centric computations in a breadth-first search matter. Ligra does no work in pre-partitioning the graph since it is not optimized for distributed shared memory and instead assumes that entire graphs can fit in a shared memory server. In settings that instead use distributed shared memory, applying Ligra's computation model would require constant swapping of memory pages, which would incur large amounts of network latency during computation.

GraphLab is another system for parallel processing of Graphs, that could also extend to a shared memory model. [18] GraphLab allows for a tunable consistency model (vertex-centric, edge-centric or full) while performing computations and does allow graph partitioning with ParMetis, the same library used by BlueBridge. However, GraphLab is a system with much more infrastructure, reading input files from a distributed storage system such as HDFS and compressing these partitions into binary files called atoms. GraphLab provides strong consistency guarantees and scales well but contains far too much overhead for the everyday scientist.

Piccolo is a distributed shared memory system that also performs pre-partitioning of data with respect to locality. Piccolo also uses a key-value table interface for defining access to shared memory. [22] Since Piccolo is also designed for DSM, locality is of high importance when accessing shared table entries, and Piccolo attempts to optimize the collocation of shared table entries. Unlike BlueBridge, Piccolo does not use a persistent RDD storage model, and instead relies on checkpointing of shared tables through a Chandy-Lamport distributed snapshot algorithm to provide fault tolerance.

5 Future Work

5.1 Implementing DSM

BlueBridge serves as a proof of concept for DSM, and has shown the potential merits of a DSM system for graph processing. Given the results shown in this analysis, BlueBridge should be implemented on an actual cluster, leveraging the potential of DSM. In order to do this, pages of memory in the DSM system would contain both local pages of memory, and shared global pages of memory. BlueBridge would load the external edges of the graph onto these shared global pages, and synchronization of these pages between iterations would be done by a shared state manager. One proposed implementation of this system is shown in Figure 4, in which outgoing external edge values are stored on shared pages and requested by other machines for their remote pages.

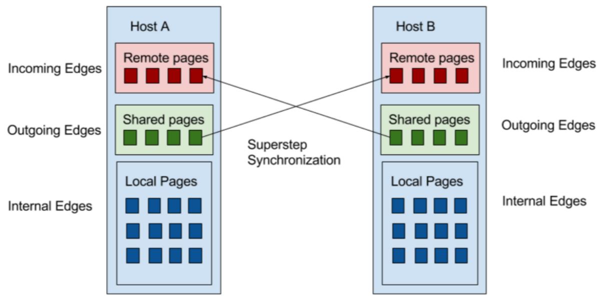

Figure 4: A proposed implementation of BlueBridge on distributed shared memory.

5.2 Offloading Work from Python

The present state of the BlueBridge prototype comes with many performance concerns. While Python provides a simple interface for the access of shared memory, use of Python's *Manager* library for access and control of shared state was unacceptable from a performance perspective. *cProfile*, the Python profiler, was used to investigate the performance of this library and it was found that up to 80-90% of the total runtime was spent in *Manager* library communication primitives, such as *send()* and *recv()*. We believe that writing a custom shared state manager to handle memory access in C would provide enormous performance benefits over the current BlueBridge prototype.

6 Conclusion

References

[1] Facebook disaggregated rack, http://goo.gl/6h2ut.

[2] Hp the machine, http://www.hpl.hp.com/research/systems-research/themachine/.

[3] Intel rsa. https://software.intel.com/en-us/articles/intel-performance-counter-monitoring.

[4] Pymetis, https://mathema.tician.de/software/pymetis/.

[5] Seamicro technology overview, http://seamicro.com/.

[6] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *Proc. VLDB Endow.*, 8(12):1804–1815, Aug. 2015.

[7] C. et al. From ”think like a vertex” to ”think like a graph”. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, 2015.

[8] T. et al. From ”think like a vertex” to ”think like a graph”. In *Proceedings of the VLDB Endowment Volume 7 Issue 3*, VLDB '14, pages 193–204, 2013.

[9] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 17–30, Berkeley, CA, USA, 2012. USENIX Association.

[10] I. F. Haddad and E. Paquin. Mosix: A cluster load-balancing solution for linux. *Linux J.*, 2001(85es), May 2001.

[11] S. Han, N. Egi, A. Panda, S. Ratnasamy, G. Shi, and S. Shenker. Network support for resource disaggregation in next-generation datacenters. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, HotNets-XII, pages 10:1–10:7, New York, NY, USA, 2013. ACM.

[12] Z. Huang, W. Chen, and et al. Vodca: View-oriented, distributed, cluster-based approach to parallel computing. In *DSM WORKSHOP 2006, IN: PROC. OF THE IEEE/ACM SYMPOSIUM ON CLUSTER COMPUTING AND GRID 2006 (CCGRID06), IEEE COMPUTER SOCIETY*, 2006.

[13] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM.

[14] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, WTEC'94, pages 10–10, Berkeley, CA, USA, 1994. USENIX Association.

[15] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 31–46, Berkeley, CA, USA, 2012. USENIX Association.

[16] D. Lasalle and G. Karypis. Multi-threaded graph partitioning. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, IPDPS '13, pages 225–236, Washington, DC, USA, 2013. IEEE Computer Society.

[17] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[18] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, Apr. 2012.

[19] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[20] C. Morin, R. Lottiaux, G. Vallee, P. Gallard, D. Margery, J.-Y. Berthou, and I. D. Scherson. Kerrighed and data parallelism: Cluster computing on single system image operating systems. In *Proceedings of the 2004 IEEE International Conference on Cluster Computing*, CLUSTER '04, pages 277–286, Washington, DC, USA, 2004. IEEE Computer Society.

[21] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 439–455, New York, NY, USA, 2013. ACM.

[22] R. Power and J. Li. Piccolo: Building fast, distributed programs with partitioned tables. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 293–306, Berkeley, CA, USA, 2010. USENIX Association.

[23] J. Shun and G. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 135–146, 2013.

[24] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 5:1–5:16, New York, NY, USA, 2013. ACM.

[25] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13, pages 2:1–2:6, New York, NY, USA, 2013. ACM.

[26] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.