Giraphx: Parallel Yet Serializable Large-Scale Graph Processing

Serafettin Tasci and Murat Demirbas

Computer Science & Engineering Department
University at Buffalo, SUNY

Abstract. Bulk Synchronous Parallelism (BSP) provides a good model for parallel processing of many large-scale graph applications, however it is unsuitable/inefficient for graph applications that require coordination, such as graph-coloring, subcoloring, and clustering. To address this problem, we present an efficient modification to the BSP model to implement serializability (sequential consistency) without reducing the highly-parallel nature of BSP. Our modification bypasses the message queues in BSP and reads directly from the worker's memory for the internal vertex executions. To ensure serializability, coordination is performed—implemented via dining philosophers or token ring— only for border vertices partitioned across workers. We implement our modifications to BSP on Giraph, an open-source clone of Google's Pregel. We show through a graph-coloring application that our modified framework, *Giraphx*, provides much better performance than implementing the application using dining-philosophers over Giraph. In fact, Giraphx outperforms Giraph even for embarrassingly parallel applications that do not require coordination, e.g., PageRank.

1 Introduction

Large-scale graph processing finds several applications in machine-learning [1], distributed simulations [2], web-search [3], and social-network analysis [4]. The significance of these applications led to the development of several graph processing frameworks recently. Due to the large size of the graphs considered, these frameworks employ a distributed/parallel execution model; Most adopt the Bulk Synchronous Parallel (BSP) [8] approach to this end. A popular example is the Pregel [3] framework from Google. Pregel inspired several open-source projects, including Apache Giraph [5], Hama [6], and Golden Orb [7], all of which use the BSP model. Although asynchronous graph processing frameworks such as GraphLab [1] and PowerGraph [16] were proposed recently, the BSP model is still used the most due to its simplicity, flexibility, and ease of use.

In the BSP approach to graph processing, the large input graph is partitioned to the worker machines, and each worker becomes responsible for execution of the vertices that are assigned to it. Then BSP's superstep concept is used for coordinating the parallel execution of the workers. A superstep consists of three parts. *Concurrent computation:* Concurrently every participating worker executes computations for the vertices they are responsible for. *Communication:*

F. Wolf, B. Mohr, and D. an Mey (Eds.): Euro-Par 2013, LNCS 8097, pp. 458–469, 2013.
© Springer-Verlag Berlin Heidelberg 2013

The workers send messages on behalf of the vertices they are responsible for to their neighboring vertices. The neighboring vertices may or may not be in the same worker. *Barrier synchronization:* When a worker reaches this point (the barrier), it waits until all other workers have finished their communication actions, before the system as a whole can move to the next superstep. A computation involves many supersteps executed one after the other in this manner. So, in a superstep, the worker uses values communicated via messages from the previous superstep, instead of most recent values.

BSP provides good parallelism and yield. However, the *max-parallel* execution model used in BSP is not suitable for writing applications that require coordination between vertices. Consider the graph coloring problem, where the aim is to find a minimal coloring of the vertices such that no two adjacent vertices share the same color. A simple program is that, at each superstep a vertex picks the smallest color not used by any of its neighbors and adopts it as its color. If executed in a max-parallel manner, this program does not converge: If two neighboring vertices have the same color at any superstep, they loop on back and forth changing their colors to be the same. This max-parallel style concurrency violations can occur even when the worker has a single thread of control[1], because vertices in a worker communicate via message-passing in queues, and as such they operate on each other's previous round states. So, in effect, all vertices in a worker are executing concurrently for a superstep, even though in reality the vertices execute in a serial manner since the worker has a single thread.

For developing applications that require coordination, the *serializability* semantics [9] (also known as *interleaving* or *sequential consistency*) is better. Serializability ensures that for every parallel execution, there exists a sequential execution that produces an equivalent result. This model provides "in effect" the guarantee that any vertex computation is executed atomically (or in isolation) with respect to the rest of the system and this gives a cleaner abstraction to write graph programs.

The problem with serializability, however, is that it may throttle the parallelism/performance of the system, so how serializability is implemented matters. The straightforward implementation (executing all vertex computations sequentially in the graph and disallowing any parallel execution across workers) is, of course, very inefficient. It is easy to observe that if two vertices are not neighbors they can be executed in parallel, and since they do not share state, their executions do not conflict with each other, and they can be serialized (pretending as if one occurs before the other). One can implement this restriction using a dining philosopher program [10] to regulate that no two neighboring nodes execute at the same superstep. Running the application on top of dining philosophers in Giraph accomplishes serializability, but with a steep cost (as we show in our experimental results in Section 4).

Our Contributions. We present a simple extension to achieve serializability in BSP-based frameworks while keeping the highly-parallel and bulk-efficient

[1] This is the case in Giraph. Even when the worker is executed on a multicore machine, the worker executes as a single thread to keep concurrency issues manageable.

nature of BSP executions. We implement our extension on the open-source Apache Giraph framework, and call the extended framework *Giraphx*. To provide interworker serializability, we augmented Giraphx with two alternative coordination mechanisms: dining philosophers and token ring.

We give experimental results from Amazon Web Services (AWS) Elastic Compute Cloud (EC2) with upto 32 workers comparing the performance of Giraphx with that of Giraph. We show through a graph-coloring application that Giraphx consistently provides better performance than implementing the application using dining-philosophers over Giraph. Our experiments use mesh graphs and Google Web graphs, and show the effects of edge-locality in improved performance (in some cases upto an order of magnitude improvement) of Giraphx compared to Giraph. The results reveal that while dining-philosopher-based Giraphx performs better for large worker numbers, the token-ring-based Giraphx is superior for smaller clusters and low edge-locality situations.

Our experiments also show that Giraphx provides better performance than Giraph even for applications that are embarassingly parallel and do not require coordination. We show this through running a PageRank [11] application on both Giraph and Giraphx. The improved performance in Giraphx is due to the faster convergence it achieves by providing the vertices the ability to read the most recent data of other vertices in the serializability model.

Overview of our Method. In Giraphx, we categorize vertices as border vertices and internal vertices. If all the neighbors of a vertex are in the same worker as that vertex, then it is an *internal vertex*; else it is called a *border vertex*. In order to provide serializability, we modify Giraph to bypass the message queue and read directly from worker's memory for the internal vertex executions (we can have direct memory reads because the vertices are in the same worker). Since vertices read current values of neighbors (instead of using previous round values from the messages), interleaving execution, and hence atomic execution is achieved. In the above example, this modification solves the graph coloring problem easily and efficiently (without being hampered by running dining philosophers on vertices and slowing execution). When border vertices, partitioned across workers, are involved, additional synchronization is needed. For this, we use dining-philosopher or a worker-based token ring algorithm for synchronizing execution. Giraphx is much cheaper than running dining philosophers over Giraph because dining philosophers is run only on cross-worker edges of border vertices (which is generally a small fraction of all the vertices) in Giraphx, so the overhead comes only on this fraction not on the entire graph as in Giraph.

Outline of the Rest of the Paper. We discuss Giraph and dining philosophers implementation on Giraph in Section 2. In Section 3, we present Giraphx, and introduce our dining-philosopher-based Giraphx (d-Giraphx) and token-ring-based Giraphx (t-Giraphx). We present our experiment results from EC2 deployments in Section 4, and related work in Section 5.

2 Giraph

Giraph leverages Hadoop's MapReduce framework [12]. The master and all workers in Giraph are started as MapReduce worker tasks. [2] Hadoop's master-worker setup readily solves the monitoring/handling reliability of the workers, optimizing performance for communication, deploying distributed code, distributing data to the workers, and load-balancing.

Writing a Giraph program involves subclassing the BasicVertex class and overriding the Compute() method. Each worker goes over the graph vertices in its assigned partitions and runs Compute() for each active vertex once in every superstep. At each Compute operation, the vertex can read the messages in its incoming queue, perform computations to update its value, and send messages to its outgoing edges for evaluation in the next superstep. A Giraph program terminates when there are no messages in transit and all vertices vote to halt.

2.1 d-Giraph

While Giraph fits the bill for many graph-processing applications, it fails to provide a mechanism for applications where neighboring vertices need to coordinate their executions. Also, while the synchronous execution model in Giraph is easy to use, the inability to read the most recent data may lead to slow convergence. Consider the graph coloring problem. If two neighboring vertices have the same color at any superstep, they loop on back and forth changing their colors to be the same.

To solve this problem, we need to schedule the computation of vertices such that no conflicting vertices operate at the same superstep. For this purpose, we developed a serializable Giraph implementation called *d-Giraph*, that ensures that in each neighborhood only one vertex can compute at a superstep while others have to wait for their turn. d-Giraph uses the hygienic dining philosophers algorithm for vertex coordination [10]. The basic d-Giraph operation consists of the following steps:

1. At superstep 0, every vertex sends a message containing its id to all outgoing edges so that at superstep 1 vertices will also learn their incoming edges.
2. At superstep 1, every vertex sends its randomly generated initial fork acquisition priority to all edges together with its vertex value for initial distribution of forks in a deadlock-free manner.
3. Computation starts at superstep 2 in which every vertex gathers its initial forks and learns initial values of neighbors.
4. Then each vertex checks if it has all its forks. If so, it performs vertex computation, otherwise it executes a skip (state is not changed).
5. Each vertex replies incoming fork request messages, and then sends request messages for its missing forks. New vertex value is sent only if it is updated.

[2] Giraph does not have a reduce phase: It uses only the map phase of MapReduce and this single map phase runs until all supersteps are completed.

Despite achieving serializability, d-Giraph hurts parallelism significantly by allowing only a small fraction of all vertices to operate at each superstep; a steep price to pay for serializability.

3 Giraphx

In order to provide efficient serializability, in Giraphx we modify Giraph to bypass the message queue and read directly from worker's memory for the internal vertex executions (we can have direct memory reads because the vertices are in the same worker). Since vertices read current values of other vertices' variables (instead of using previous round values from the messages), interleaving execution, and hence atomic execution is achieved. When border vertices, partitioned across workers, are involved, additional synchronization is needed. A border vertex cannot make direct memory reads for its interworker edges and blocking remote reads are costly. In this case, we revert to the BSP model and use messaging in tandem with a coordination mechanism for these border edges. We propose two such mechanisms: a dining philosopher based solution as in Section 2.1 called *d-Giraphx* and a simpler token-based solution called *t-Giraphx*.

The only side effect of these modifications to Giraph semantics is the increase in update frequency of internal vertices compared to border vertices. However this difference causes no complications in most graph problems.

To ease migration from Giraph, Giraphx closely follows the Giraph API except small modifications in handling of intra-worker communications. To implement Giraphx, we added approximately 800 lines of Java code to Giraph (including the coordination mechanisms). While T-Giraphx has no memory overhead over Giraph, d-Giraphx uses $\sim 30\%$ more memory primarily for synchronization messages (i.e. fork exchange) in dining philosophers.

3.1 d-Giraphx

d-Giraphx uses dining philosophers for establishing coordination of the interworker edges in Giraphx. To implement d-Giraphx, d-Giraph is modified as follows:

1. Each vertex prepares a list that denotes the locality information about the neighbors. If all neighbors are local then the vertex marks itself as *internal*, else as *border*.
2. If a vertex is internal, it operates at each superstep. If it is a border vertex, it checks whether it has gathered all forks or not. If yes, it first iterates over local neighbor list and reads their values and then iterates over its incoming messages to learn the values of its nonlocal neighbors.
3. Border vertices are also responsible for sending and replying fork exchange messages while internal vertices skip it.

Since the amount of messaging in d-Giraphx is proportional to the number of interworker edges and border vertices, partitioning plays an important role in

improving the performance of the system. In a smartly-partitioned graph, since the border nodes is a small fraction of the internal nodes, d-Giraphx performs upto an order of magnitude better than d-Giraph as we show in Section 4.

3.2 t-Giraphx

Despite d-Giraphx is an obvious improvement over d-Giraph, it suffers from large coordination overhead when the number of border nodes is large (i.e., the edge-locality is low). To address these low edge-locality situations for which any smart-partitioning of the input graph does not provide much benefit, a solution which avoids coordination messages would be preferred. For this purpose, we implement a token-based version of Giraphx called t-Giraphx.

In t-Graphx, coordination is done at the worker level instead of at the vertex level. At each superstep one of the workers has the token for computation. When a worker acquires the token, it runs Compute() on all its vertices whether they are border or internal. Similar to d-Giraphx, vertices use messaging for inter-worker neighbors and direct memory reads for same-worker neighbors. When a worker does not have the token, it can only operate on internal vertices. In a non-token worker, the border vertices skip computation in this superstep, but they still broadcast their state to neighboring vertices in other workers by sending a message. Since t-Giraphx does not have any fork exchange overhead, it uses less messaging and thus converges faster than d-Giraphx, whenever the number of workers is sufficiently small.

t-Giraphx does not scale as the number of workers increase. Consider a graph problem where convergence is acquired after k iterations at each vertex. In a graph with average vertex degree w, d-Giraphx processes all vertices once in approximately every w supersteps independent of worker number N, and thus converges in at most $w * k$ supersteps independent of N. On the other hand, t-Giraphx will need $N * k$ supersteps, resulting in longer completion times when $N >> w$, and shorter completion times when $N << w$.

Some problems, such as graph subcoloring[3], require multi-hop coordination. While dining philosophers achieves 1-hop neighborhood coordination, it can be extended to provide multi-hop neighborhood coordination by defining the multi-hop neighbors as “virtual” neighbors. For example, for 2-hop coordination, this can be done by adding another superstep in d-Giraphx after superstep 0 in which vertices send a message containing the ids of all neighbors to every neighbor. This way, in the following superstep all vertices will have a list of vertices in its 2-hop neighborhood and then can run dining philosophers on this list. The primary drawback of these multihop coordination extensions is the significant increase in the number of edges per vertex, and the resulting increase in the total running time of the protocol. In applications that require multihop coordination, since w will grow exponentially, t-Giraphx becomes a better alternative to d-Giraphx.

[3] In graph subcoloring, the aim is to assign colors to a graph's vertices such that each color class induces a vertex disjoint union of cliques. This requires 2-hop neighborhood coordination.

4 Experiments

To evaluate the performance of Giraphx, we conducted experiments on EC2 using medium Linux instances, where each instance has two EC2 compute units and 3.75 GB of RAM. In our experiments, we used up to 33 instances, where one instance is designated as the master, and the remaining 32 instances are workers. We used two network topologies for our experiments. First is a planar mesh network where each node is connected to its 4 neighbors in the mesh (right, left, top, bottom) plus one of the diagonal neighbors (e.g. right-top). The second is Google's web graph dataset [14] in which vertices represent web pages and directed edges represent hyperlinks between them. The Google web graph dataset consists of approximately 900,000 vertices and 5 million edges. This dataset is quite challenging since it has a highly skewed degree distribution where some vertices have degrees up to 6500.

We used three partitioners in our experiments: a hash partitioner (which assigns vertices to workers pseudo-randomly), a mesh partitioner (which assigns each worker a neighborhood in the mesh), and the metis partitioner (which smartly-partitions a graph to provide high edge-locality) [13]. For Giraphx, the cost of learning internal versus border vertices are included in the provided experiment results. In all experiments, the partitioning run times are included in the results.

4.1 Graph-Coloring Experiments on Mesh Graph

We first fix the number of workers as 16, and perform experiments with increasing the number of vertices in a mesh graph. We use a mesh-partitioner so the percentage of local vertices in these experiments stays between 94%–99%. Figure 1 demonstrates that as the size of the graph increases the running time of d-Giraph increases at a much faster pace than Giraphx-based methods. The basic reason is the lack of d-Giraph's ability to make use of the locality in the graph. Every vertex needs to coordinate with all neighbors causing large delays. While the local nodes in Giraphx can compute at every superstep, in d-Giraph vertices have to wait until all forks are acquired to make computation. In contrast, d-Giraphx avoids a significant fraction of the messaging in d-Giraph by incorporating local memory reads for internal vertices.

Figure 1(b)) shows that increase in the graph size does not necessarily increase the number of supersteps for the three frameworks compared. Since the graph topology does not change, average vertex degree and hence the number of supersteps is stable as the graph size changes in d-Giraph and d-Giraphx. Since worker number does not change, superstep number in t-Giraphx is also stable. We also see that t-Giraphx takes more supersteps than the other methods, since it takes 16 supersteps for the token to return to a worker. However, t-Giraphx compensates this disadvantage and converges in less time than d-Giraph by avoiding the coordination delays that dining-philosophers induce for d-Giraph.

Next in Figure 2 we fix the number of vertices as 1 million, and increase the number of workers to observe the effects of parallelism on the runtime. As long

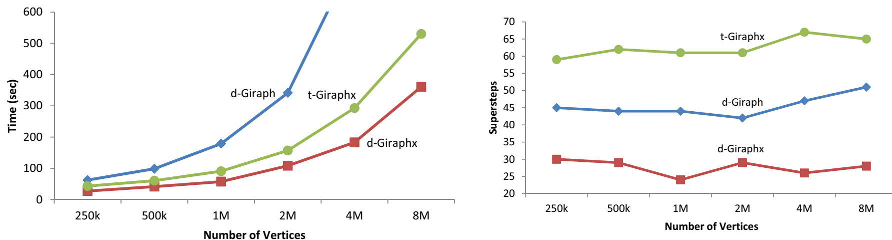

Fig. 1. Change in time and superstep number as the graph size increases for 16 workers

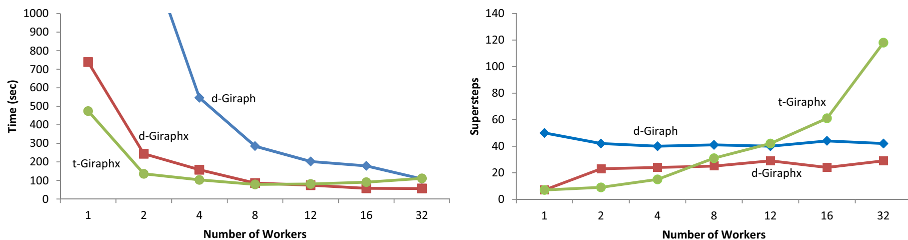

Fig. 2. Change in time and superstep number as the amount of worker machines increases for 1 million vertices

as the edge-locality is high, d-Giraph and d-Giraphx take advantage of every worker added since the load per worker decreases accordingly. t-Giraphx also benefits from the increase in computation power. But after 8 workers, adding more workers hurts t-Giraphx due to the increase in the superstep numbers proportional to the number of workers.

Finally, in Figure 3, we ran an experiment where we use the hash partitioner on the mesh network to show the effects of partitioning. While the mesh partitioner provides an optimal partitioning in a mesh network, using a hash partitioner causes vertices to lose their locality and most of them become border vertices. As a result, d-Giraphx loses the performance improvement it gains from internal vertices, and performs only slightly better than d-Giraph. The performance of t-Giraphx is not affected too much, since it does not suffer from the increased coordination cost on border vertices.

4.2 Graph-Coloring Experiments on Google Web Graph

To reveal the performance of the proposed frameworks on a real-world graph, we ran another set of experiments on the challenging Google web graph. This graph has a highly skewed degree distribution and it is hard to find a partitioning that will provide good edge-locality. These high-degree vertices cause communication and storage imbalance on vertices as well as imperfect partitioning. In this graph, using the hash partitioner 99% of all vertices become border

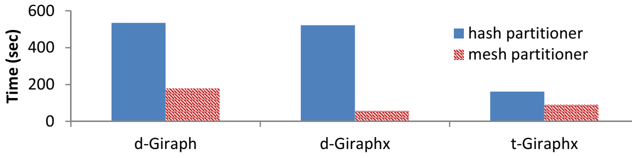

Fig. 3. Computation times with hash versus mesh partitioners for 16 workers and 1M vertices

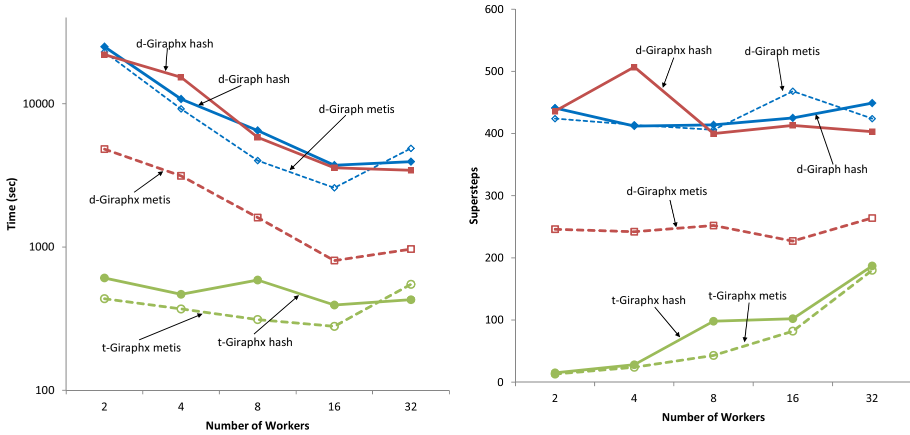

Fig. 4. Results with hash versus metis partitioner on the web graph. Solid lines indicate hash partitioner, and dashed lines metis partitioner. Time is given in log scale.

vertices, and this hurts parallelism immensely. To prevent the loss of parallelism and improve the locality, we preprocessed the graph using the metis partitioner. When metis is used, ratio of border vertices drops to around 6%. However these vertices are mostly high-degree hubs which have a lot of interworker neighbors. Therefore even with metis, webgraph still performs worse than the same-sized mesh-partitioned mesh network.

Figure 4(a) shows a comparison of the three frameworks on the Google web graph as the number of workers increase. Regardless of whether metis is used or not, t-Giraphx always performs better than other methods. This is because t-Giraphx has a predictable number of supersteps independent from the degree distribution while the number of high-degree vertices adversely affect the number of supersteps in d-Giraph and d-Giraphx (see Figure 4(b)).[4] The skewed degree distribution leads to a star topology centered on high-degree vertices resulting in larger fork re-acquisition intervals for d-Giraph and also for d-Giraphx to a

[4] In the mesh graph where the maximum degree is 5, the number of supersteps required for all vertices to compute at least once is around 10. In comparison, in the web graph where the maximum degree is 6000 the number of supersteps jumps to 50 for metis partitioning and 70 for hash partitioning.

degree. d-Giraphx improves with better partitioning in terms of both time and supersteps, because even the highest degree vertices now have a much higher number of local neighbors decreasing the size of the subgraph on which coordination through fork exchange is required.

An interesting result is the increase in runtime as the number of workers exceeds 16 in metis partitioning. In t-Giraphx, the increase in the number of supersteps is the dominant reason for this. In case of d-Giraph and d-Giraphx, after 16 workers the computation power gained by adding new workers is overshadowed by the loss of locality.

4.3 Pagerank Experiments on Google Web Graph

Giraphx also provides performance improvements for graph applications that do not require coordination among vertices. To demonstrate this, we modified the PageRank implementation in the Giraph framework, and ran it on Giraphx to compare their performances. PageRank computes the importance of the vertices/webpages iteratively using a weighted sum of neighbor values, and does not require serializability among vertex computations.

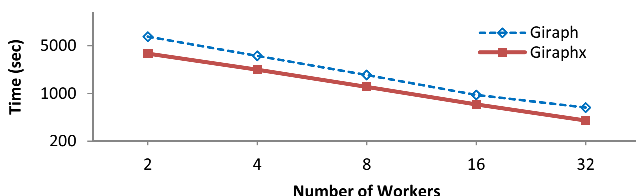

Fig. 5. Comparison of Giraph and Giraphx on Pagerank application (log scale)

In this experiment we used the Google web graph with metis partitioning. In our implementation, vertices vote for termination when $\Delta < 10^{-9}$ where Δ is the total change in vertex values from the previous superstep. Figure 5 shows that Giraphx finishes computation %35 faster than Giraph, for all worker numbers. The improved performance in Giraphx is due to the faster convergence it achieves. The experiment logs reveal that convergence takes just 64 supersteps in Giraphx compared to 117 supersteps in Giraph.

In Giraphx, the internal vertices take advantage of the direct memory reading feature, and read the most recent data of other vertices, which leads to faster convergence times. In fact, the same argument is also cited as the main advantage of asynchronous systems (e.g. GraphLab [1]) over the BSP model as we discuss in Section 5. Giraphx can provide faster convergence as in asynchronous systems without sacrificing the convenience of the BSP model for application development and debugging.

5 Related Work

The alternative approach to synchronous BSP-based systems is to use an asynchronous system which updates the vertices immediately and therefore uses the most recent data at any point in computation. In addition, asynchronous systems avoid the barrier synchronization cost. However asynchrony brings along programming complexity and may require consideration of consistency issues.

Distributed GraphLab [1] is a well-known powerful and flexible asynchronous framework. Asynchrony helps faster convergence and vertex prioritization but it requires selection of a consistency model to maintain correct execution in different problems. In addition, unlike GiraphX, it does not allow dynamic modifications to graph structure. PowerGraph [16] proposes a unified abstraction called Gather-Apply-Scatter which can simulate both asynchronous and synchronous systems by factoring vertex-programs over edges. This factorization helps reduction of communication cost and computation imbalance in power-law graphs. However, it has the same shortcomings as GraphLab. GRACE [15] also provides a parallel execution engine which allows usage of both execution policies.

Since partitioning plays an important role in efficient placement of graph data over cluster nodes, some studies focus on partitioning the graph data. A recent work [18] shows that SPARQL queries can be processed up to 1000 times faster on a Hadoop cluster by using a clever partitioning, custom data replication and an efficient data store optimized for graph data. A bandwidth aware graph partitioning framework to minimize the network traffic in partitioning and processing is proposed in [19]. Finally, another recent work [20] shows that using simple partitioning heuristics can bring a significant performance improvement that surpasses the widely-used offline metis partitioner.

6 Conclusion

In this paper, we proposed efficient methods to bring serialization to the BSP model without changing its highly-parallel nature and clean semantics. We showed how dining philosophers and token ring can be used for achieving coordination between cross-worker vertices. We alleviated the cost of these coordination mechanisms by enabling direct memory reads for intraworker vertex communication.

We implemented Giraphx on top of Apache Giraph and evaluated it on two applications using real and synthetic graphs. We showed through a greedy graph coloring application that Giraphx achieves serialized execution in BSP model with consistently better performances than Giraph. Our experiment results showed that while d-Giraphx performs better for large worker numbers, t-Giraphx performs better for small worker numbers and low edge-locality situations. Finally we showed that, due to the faster convergence it provides, Giraphx outperforms Giraph even for embarrassingly parallel applications that do not require coordination, such as PageRank.

References

1. Low, Y., Bickson, D., Gonzalez, J., Guestrin, C., Kyrola, A., Hellerstein, J.M.: Distributed graphlab: a framework for machine learning and data mining in the cloud. Proc. VLDB Endow. 5(8), 716–727 (2012)
2. Braun, S.A.: A cloud-resolving simulation of hurricane bob (1991): Storm structure and eyewall buoyancy. Mon. Wea. Rev. 130(6), 1573–1592 (2002)
3. Malewicz, G., Austern, M.H., Bik, A.J., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: a system for large-scale graph processing. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, pp. 135–146. ACM, New York (2010)
4. http://www.facebook.com/about/graphsearch/
5. http://incubator.apache.org/giraph/
6. http://hama.apache.org/
7. http://goldenorbos.org/
8. Valiant, L.G.: A bridging model for parallel computation. Commun. ACM 33(8), 103–111 (1990)
9. Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann Publishers (1993)
10. Chandy, K.M., Misra, J.: The drinking philosopher's problem. ACM Trans. Program. Lang. Syst. 6(4), 632–646 (1984)
11. Brin, S., Page, L.: The anatomy of a large-scale hypertextual web search engine. In: Proceedings of the Seventh International Conference on World Wide Web 7, WWW7, pp. 107–117 (1998)
12. http://hadoop.apache.org/
13. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM Journal on Scientific Computing 20(1), 359 (1999)
14. http://snap.stanford.edu/data/web-Google.html/
15. Wang, G., Xie, W., Demers, A., Gehrke, J.: Asynchronous large-scale graph processing made easy. In: Proceedings of CIDR 2013 (2013)
16. Gonzalez, J.E., Low, Y., Gu, H., Bickson, D., Guestrin, C.: Powergraph: Distributed graph-parallel computation on natural graphs. In: Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood (October 2012)
17. Kyrola, A., Blelloch, G., Guestrin, C.: Graphchi: Large-scale graph computation on just a pc. In: Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood (October 2012)
18. Huang, J., Abadi, D.J., Ren, K.: Scalable sparql querying of large rdf graphs. PVLDB 4(11), 1123–1134 (2011)
19. Chen, R., Yang, M., Weng, X., Choi, B., He, B., Li, X.: Improving large graph processing on partitioned graphs in the cloud. In: Proceedings of the Third ACM Symposium on Cloud Computing, SoCC 2012, pp. 3:1–3:13. ACM, New York (2012)
20. Stanton, I., Kliot, G.: Streaming graph partitioning for large distributed graphs. In: Yang, Q., Agarwal, D., Pei, J. (eds.) KDD, pp. 1222–1230. ACM (2012)