Horizon: A Multi-abstraction Framework for Graph Analytics

Adnan Haider*
Illinois Institute of Technology
ahaider3@hawk.iit.edu

Fabio Checconi
IBM Research
fchecco@us.ibm.com

Xinyu Que
IBM Research
xque@us.ibm.com

Lars Schneidenbach
IBM Research
schneidenbach@us.ibm.com

Daniele Buono
IBM Research
dbuono@us.ibm.com

Xian-He Sun
Illinois Institute of Technology
sun@iit.edu

ABSTRACT

A graph application written using a distributed graph processing framework can perform over an order of magnitude slower than its high-performance, native counterpart. This issue stems from the aim, common to most graph frameworks, of restricting the scope of application development to specific graph constructs, such as, for example, vertex or edge programs.

In this paper we present Horizon, a distributed graph processing framework achieving close to native performance without penalizing productivity by providing a multi-layer, multi-abstraction model of computation. Compared to current frameworks, Horizon extends the scope of computation by exposing two notions usually relegated to implementations: graph data models and communication models. Horizon can reduce execution time by an average of 5.3× across different applications and datasets and process an order of magnitude larger graphs when compared to the state of the art.

ACM Reference Format:
Adnan Haider, Fabio Checconi, Xinyu Que, Lars Schneidenbach, Daniele Buono, and Xian-He Sun. 2018. Horizon: A Multi-abstraction Framework for Graph Analytics. In *CF '18: CF '18: Computing Frontiers Conference, May 8–10, 2018, Ischia, Italy.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3203217.3203270

1 INTRODUCTION

Graph algorithms have found application in a variety of scientific and business domains, and are employed in fields such as bioinformatics and machine learning [6]. Yet, the research community has been struggling to find the definition of a programming model capable of describing graph algorithms in a way that is both easy to program and efficient to execute. As part of this effort, many distributed graph frameworks have been developed [2–4, 8, 10]. These frameworks were designed to provide users with a productive method to implement graph algorithms on distributed systems, with the proposed programming models offering abstractions such as vertices, edges, blocks, matrices, or domain-specific languages to reduce the amount of effort required by users to develop their applications.

*in IBM Research at the time.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CF '18, May 8–10, 2018, Ischia, Italy
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5761-6/18/05…$15.00
https://doi.org/10.1145/3203217.3203270

For example, Combinatorial BLAS (CombBLAS) [2] is a distributed graph processing framework based on linear algebra primitives. Users express sparse matrix and vector operations via different semirings. GraphX [3] is an embedded graph processing framework built on top of Apache Spark [9], that allows the user to process graphs in an interactive, distributed manner.

Graph processing frameworks, however, generally fail to approach the performance provided by native, hand-optimized code [5, 8]. The authors of [5] showed that many of these frameworks, running on small-size clusters, can be outperformed by single-threaded native code running on a laptop. The adoption of techniques generally used in native code inside the implementation of a framework's backend has been proposed in several instances; Gemini [10] is worth noting since it presents an optimized distributed graph processing system that contribute to making it one of the best performing frameworks currently available. However, the number of graph frameworks available demonstrates that no single one of them is able to provide performance comparable to native code for all input datasets, algorithms, and hardware variations. We believe the reason is that most graph frameworks operate at a level of abstraction that hides performance-critical primitives from the user in order to make applications easy to develop.

In this paper we present Horizon, a new framework to bridge the gap between programmability and performance. Horizon presents the programmer multiple levels of abstraction. This allows the programmer to select the abstraction that better fit each part of the application, without giving up on programmability. When required, specific parts of the application can be developed using a lower level of abstraction, one that provides the programmer with more tools to optimize the specific part and reach nearly-native performance.

The additional benefit given by the presence of multiple levels of abstraction is that it enables the user to specialize the application incrementally: a developer can start with an initial implementation entirely written using a high-level vertex-centric programming model—as they would do in a classic graph framework—and then, profiling the resulting code, they can identify which parts of the application are performance-sensitive and specialize only those parts. Our contributions include:

- The design of an API that abstracts optimizations typical of native codem and its implementation in Horizon, showing it can reduce runtime on average 5.3× across different graph algorithms and datasets, and solve problems that are an order of magnitude larger, compared to the state of the art graph frameworks (Section 4).
- The implementation of four well-known graph analytics benchmarks, to show the capabilities of Horizon both in terms of expressiveness and performance.

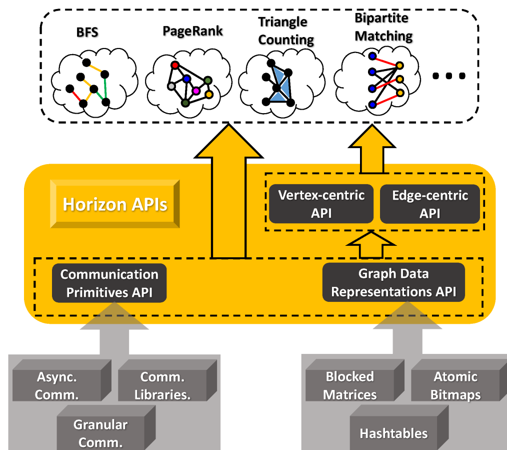


Figure 1: Overview of Horizon

2 HORIZON DESIGN

Horizon is, by design, a collection of components that allow users to develop graph applications. We define Horizon's API as an object oriented C++ interface. The components provided by Horizon can be grouped into two categories: *data model* and *communication*, as depicted in Fig. 1. We incorporated into the API data structures and communication methods frequently encountered in native code, and we defined the object interfaces in a way that allows high-performance implementations (e.g., allowing asynchronous messaging, in order to overlap computation and communication). Such components provide the lower level of Horizon. To provide flexibility, they do not adhere to a specific programming model, and do not offer concepts commonly found in current programing models. On top of such API, Horizon provides *vertex-centric* and *edge-centric* APIs. To provide a high level of interchangeability, such APIs are implemented using the components of the "low-level". This common base allows the developer to mix different APIs within the same program, and to use the low-level of Horizon only when deemed necessary.

3 APPLICATION DEVELOPMENT IN HORIZON

In Section 4 we will compare Horizon and a set of over a set of four algorithms. Because of the level of optimization required, the algorithms presented will exploit the low-level API offered by Horizon. *This is, however, solely because of performance requirements.* To highlight that, we compare, in Table 1, the source lines of code (SLOC) required to implement all of the algorithm using both the vertex-centric and the low-level interfaces of Horizon.

For the sake of simplicity we will present single-threaded distributed code. While this can still exploit multicore architectures by just running multiple processes within the node, the codes analyzed in the experiments section employ thread-level parallelism inside each node.

3.1 Breadth-first Search: a case study

To offer a brief idea of how algorithms can be implemented in Horizon, we discuss a possible implementation of BFS. We will focus on implementing the Direction-optimized[1] variant of BFS, given the performance advantage over the classic approach. In order to support processing with in-edges and out-edges, we use two different representations of the same graph, and count the active vertices to select which visit to trigger at each BFS iteration. The pseudocode of the algorithm is shown in Figure 2. The *main* function initializes the representations, the communicators and the auxiliary data structures (two bitmaps), and then starts computing BFS. The main loop is executed until we don't have active edges (i.e. the whole graph has been traversed). For each iteration, depending on the set of active vertices, we trigger a visit on one of the two representations. The *sync* call is used to guarantee that all the messages have been sent and processed, and to compute the global number of active vertices. *popcnt* is a generalization of the well-known assembler instruction and counts the number of bits set to 1 in the bitmap. In addition to the main function, we need two other functions, *send_edge* which is called by the foreach construct, and send a message containing the edge to the node with the destination vertex, and *process_edge* which is the callback invoked every time a message is received, and is in charge of updating the local bitmaps.

In order to improve cache locality, both representations (*csr_out* and *csr_in*) are blocked matrices. Users can specify the dimensions of the matrix to make block sizes fit in the last level cache for performance tuning. A further optimization is introduced by using a bitmap communicator, to compress outgoing messages.

Application	Gemini	Horizon vertex-centric	Horizon low-level
Breadth-first Search (BFS)	81	30	145
Pagerank	100	22	118
Triangle Counting	72	29	126
Bipartite Matching	77	92	42

Table 1: Number of source lines of code (SLOC) across different frameworks and apps

```
void process_edge(msg m)
  // If the node was not yet visited, mark it visited and active
  if(!visited_bitmap.get(local(m.src)))
    active_bitmap.set(local(m.src));
    visited_bitmap.set(local(m.src));
void send_edge(edge e)
  // Send the edge to the destination node
  msg m = { .dst = e.dst_id, .src= e.src_id };
  comm.add_msg(m, comm.get_owner(e.dst_id));
int main()
  csr_out = new bcsr(graph_path, in_edges=0,
                     block_dim=bdim);
  csr_in  = new bcsr(graph_path, in_edges=1,
                     block_dim=bdim);
  active_bitmap = new bitmap(num_local_vertices,0);
  visited_bitmap = new bitmap(num_local_vertices,0);
  comm = new bitmap_comm();
  comm.set_handler(process_edge);
  active = 1;
  while(active > 0)
    // Select the visit depending on the threshold
    if (active < THRESHOLD)
      foreach_edge(csr_out, send_edge,
                   vertex_filter = active_bitmap);
    else
      foreach_edge(csr_in,  send_edge,
                   vertex_filter = active_bitmap);
    // Reduce to get global active count
    active = comm.sync(popcnt(active_bitmap));
```

Figure 2: Pseudo-code of DOBFS in Horizon

4 EXPERIMENTS

In this section we compare Horizon, against state-of-the-art graph frameworks. We continue to use the four benchmarks introduced

Dataset	Vertices	Edges
Livejournal	4M	68M
Twitter	41M	1,4B
Friendster	65M	1.8B
RMAT-24	16M	536M
RMAT-26	67M	2.1B
RMAT-27	134M	4.3B
RMAT-28	268M	8.6B
RMAT-29	536M	17.2B
bm-24	34M	1B
bm-25	67M	2.1B

Table 2: Graph Datasets

in Section 3, so that the reader is already familiar with the range of optimizations required by the algorithms. We also discuss the performance impact of using specific optimizations on BFS, PageRank and BM.

Graph Algorithms and Datasets. Table 2 shows the graphs we used in our evaluation: we consider both social networks and synthetic graphs. RMAT is a synthetic graph model used to replicate the characteristics of social graphs in benchmark environments (e.g., Graph500). Bipartite Matching uses a different set of synthetic graphs because of the requirements from the algorithm. With iterative algorithms (BFS and PR) we execute the first ten iterations; the other algorithms are run until completion.

Graph Frameworks. We compared against three different graph processing frameworks: GraphX, Gemini, and CombBlas. GraphX is based on Spark [3] and is known to perform poorly compared to other frameworks [5]. Gemini is a vertex-centric framework that significantly outperforms other vertex-centric graph processing [10]. To the best of our knowledge it is the *fastest vertex-centric graph processing framework*. CombBlas stand up by offering a linear algebra abstraction to graph analytics. In cases where linear algebra maps efficiently to the problem, it has been shown to perform faster than other graph frameworks [2]. All graph frameworks we used for comparison are open-source.

Comparison Method. In order to provide a fair comparison with other frameworks, for each algorithm we use the implementation provided by the authors when available. However, for some cases such as triangle counting in CombBlas an open-source implementation is not avaiable. In these cases, we either use an implementation based on pseudocode of previous work [7], or avoid the specific comparison. In addition, to keep a fair comparison with other frameworks, Horizon avoids the usage of any hardware specialized communication library such as RDMA.

Testbed. We conducted our experiments on a cluster consisting of 8 IBM Power S822LC nodes. Each node consists of two POWER8 processors and 512 GB of memory. Each POWER8 has 10 cores that run at 3.424 GHz. Each core has 64 KB of data cache, 32 KB of instruction cache, 512 KB of L2 cache and 8 MB of L3 cache. Each core supports up to 8 hardware threads. The power consumption of each chip is 190 W. The network interconnect is Infiniband Connect-X 4x EDR. Each node has a raw injection bandwidth of 12.5GB/s. The operating system is Red Hat Enterprise Linux Server version 7.3 Maipo. IBM XL C/C++ version 13.1.4 was used for compilation.

Graph	GraphX	CombBlas	Gemini	Horizon
BFS				
Twitter	545.74	1.92	0.28	0.17
Friendster	1981.19	2.97	0.41	0.37
RMAT-26	533.34	0.40	0.34	0.07
RMAT-27	-	0.81	0.43	0.20
RMAT-28	-	1.55	1.04	0.31
RMAT-29	-	3.10	1.81	0.56
Pagerank				
Twitter	360.87	19.90	1.82	1.51
Friendster	563.39	36.58	4.67	1.84
RMAT-26	611.47	4.38	2.81	2.36
RMAT-27	-	10.67	6.18	4.29
RMAT-28	-	24.74	13.45	8.35
RMAT-29	-	58.44	30.92	15.53
Triangle Counting				
Livejournal	149.86	-	33.01	8.66
RMAT-24	394.56	-	292.69	190.81
Twitter	-	-	-	271.23
RMAT-26	-	-	-	1146.78
RMAT-27	-	-	-	2949.74
RMAT-28	-	-	-	9047.26
Bipartite Matching				
BM-24	N/A	N/A	3.65	0.26
BM-25	N/A	N/A	7.5	0.47

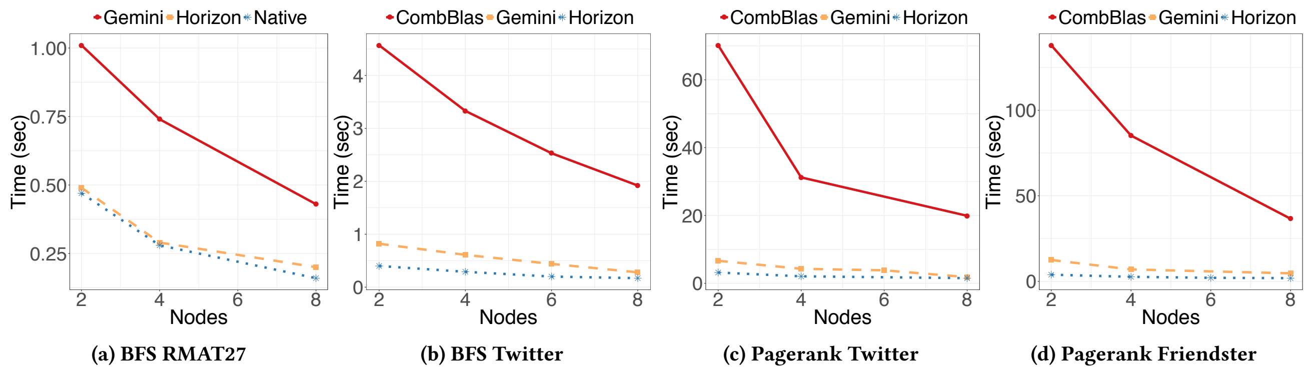
Table 3: Performance on 8 Nodes in Runtime seconds ("-" means the framework ran out of memory, N/A means the algorithm was not available for the framework)

4.1 Comparison with other frameworks

Table 3 summarize the execution runtimes for our set of algorithms, using the entire cluster (8 nodes). The results show a clear advantage of Horizon over the other frameworks in terms of performance. Specifically, Horizon performs on average 2.72× better than Gemini and 6.60× better than CombBlas on BFS. The larger speedup over CombBlas is because CombBlas does not use direction optimization; Gemini, however, does. The speedup with respect to Gemini originates from the use of blocked data structures to reduce the amount of sparse random reads of vertex states.

Horizon performs 1.67× better than Gemini and 7.35× better than CombBlas on Pagerank. The larger speedup over CombBlas is due to the use of local pagerank vectors to reduce the amount of communication. The benefits of blocked data structures vary depending on the natural locality of vertices within the graph, which is why there exists a 2.54× speedup on the friendster dataset but only a 1.21× speedup on twitter.

Triangle Counting is an example of an application that can significantly benefit from fine-grained communication by leveraging the communication model abstraction of Horizon. The advantage is most evident on the size of the datasets we can process on Horizon. Table 3 shows that Horizon allows for TC on graphs that are an order of magnitude larger than those allowed by any other framework. As vertices add their neighborhood to message buffers, Gemini runs out of memory. In CombBlas, triangle counting is expressed as a the intersection between a matrix A and its square A^2, making the algorithm even more inefficient in terms of memory requirements, as studied before in [7].



(a) BFS RMAT27 (b) BFS Twitter (c) Pagerank Twitter (d) Pagerank Friendster

Figure 3: Strong scaling of Horizon

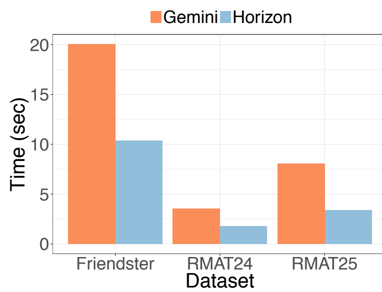


Figure 4: Single node performance of Pagerank

Since there are no official implementations of Bipartite Matching on CombBlas and GraphX, for BM we compare only with Gemini. Horizon performs 14× better than Gemini. The core issue for Gemini is that, since the scope of computation is limited to vertices and their data, the algorithm cannot exploit knowledge of distributed execution. In contrast, Horizon assigns working sets to each process so that their updates to data structures do not conflict.

4.2 Scalability

In this section, we analyze the scalability properties of Horizon, by presenting strong scaling and single node performance. For the sake of conciseness, we only report results for BFS and PR. Figure 3 shows strong scaling for different frameworks with two different data sets. Horizon outperforms all frameworks on any configurations and scales at a better rate than Gemini, although scalability is obviously not ideal. A more interesting picture emerges in Figure 3a, where the comparison includes the native code (that is only available on syntetic graphs). Horizon scales at the same rate from 2 to 4 nodes but is less effective from 4 to 8 nodes when compared to native code.

Recently, the authors of [5] highlighted a significant problem, of graph frameworks by showing that the performance of distributed graph analytics frameworks can in fact be lower than single-node shared memory implementations, because of added overheads.

To show that Horizon's performance is not significantly affected by such overheads, we also present sigle-node performance on PR. We compare with Gemini, that in turn demostrated to provide performance comparable to shared memory graph frameworks [10]. Figure 4 shows that Horizon can provide roughly 2× shorter runtime than Gemini even on a single node case.

5 CONCLUSION

In this paper, we presented Horizon, a graph processing framework extending the toolset available to the users by exposing data and communication models. Horizon provides on average 5.3× reduction in runtime as well as processing an order of magnitude larger graphs when compared to the state-of-the-art.

REFERENCES

[1] Scott Beamer, Krste Asanović, and David Patterson. 2012. Direction-optimizing Breadth-first Search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12)*. IEEE Computer Society Press, Los Alamitos, CA, USA, Article 12, 10 pages. http://dl.acm.org/citation.cfm?id=2388996.2389013

[2] Aydin Buluc and John R Gilbert. 2011. The Combinatorial BLAS: Design, Implementation, and Applications. *Int. J. High Perform. Comput. Appl.* 25, 4 (Nov. 2011), 496–509. https://doi.org/10.1177/1094342011403516

[3] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 599–613.

[4] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A System for Large-scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD '10)*. ACM, New York, NY, USA, 135–146. https://doi.org/10.1145/1807167.1807184

[5] Frank McSherry, Michael Isard, and Derek G. Murray. 2015. Scalability! But at what COST?. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. USENIX Association, Kartause Ittingen, Switzerland. https://www.usenix.org/conference/hotos15/workshop-program/presentation/mcsherry

[6] Sujith Ravi. 2016. Graph-powered Machine Learning at Google. (2016). http://arxiv.org/abs/1107.0922

[7] Nadathur Satish, Narayanan Sundaram, Md. Mostofa Ali Patwary, Jiwon Seo, Jongsoo Park, M. Amber Hassaan, Shubho Sengupta, Zhaoming Yin, and Pradeep Dubey. 2014. Navigating the Maze of Graph Analytics Frameworks Using Massive Graph Datasets. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*. ACM, New York, NY, USA, 979–990. https://doi.org/10.1145/2588555.2610518

[8] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R. Dulloor, Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. 2015. GraphMat: High Performance Graph Analytics Made Productive. *Proc. VLDB Endow.* 8, 11 (July 2015), 1214–1225. https://doi.org/10.14778/2809974.2809983

[9] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. 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)*. USENIX Association, Berkeley, CA, USA, 2–2. http://dl.acm.org/citation.cfm?id=2228298.2228301

[10] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A Computation-Centric Distributed Graph Processing System. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, GA, 301–316. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/zhu