PGX.D: A Fast Distributed Graph Processing Engine

Sungpack Hong
Oracle Labs, USA
sungpack.hong@oracle.com

Siegfried Depner
Oracle Labs, USA
siegfried.depner@oracle.com

Thomas Manhardt
Oracle Labs, USA
thomas.manhardt@oracle.com

Jan Van Der Lugt[*]
Google, USA
janlugt@gmail.com

Merijn Verstraaten[†]
Univ. Amsterdam, Netherlands
M.E.Verstraaten@uva.nl

Hassan Chafi
Oracle Labs, USA
hassan.chafi@oracle.com

ABSTRACT

Graph analysis is a powerful method in data analysis. Although several frameworks have been proposed for processing large graph instances in distributed environments, their performance is much lower than using efficient single-machine implementations provided with enough memory. In this paper, we present a fast distributed graph processing system, namely PGX.D. We show that PGX.D outperforms other distributed graph systems like GraphLab significantly (3x – 90x). Furthermore, PGX.D on 4 to 16 machines is also faster than an implementation optimized for single-machine execution. Using a fast cooperative context-switching mechanism, we implement PGX.D as a low-overhead, bandwidth-efficient communication framework that supports remote data-pulling patterns. Moreover, PGX.D achieves large traffic reduction and good workload balance by applying selective ghost nodes, edge partitioning, and edge chunking transparently to the user. Our analysis confirms that each of these features is indeed crucial for overall performance of certain kinds of graph algorithms. Finally, we advocate the use of balanced beefy clusters where the sustained random DRAM-access bandwidth in aggregate is matched with the bandwidth of the underlying interconnection fabric.

1. INTRODUCTION

Recently, graph analysis has been drawing a lot of attention from both industry and academia, as a powerful method for data analysis. In graph analysis, the underlying data set is represented as a graph such that the arbitrary, fine-grained relationships between data entities are explicitly captured as the edges in the graph. By following these edges, graph analysis naturally considers even the indirect, multi-hop relationships across the whole dataset. Consequently, graph analysis can discover valuable, non-obvious information about the original data set.

However, applying graph analysis on a very large dataset in a fast and scalable manner can be challenging. A typical large-scale graph analysis exhibits the following characteristics:

[*]This work was done during the author's employment at Oracle Labs

[†]This work was done during the author's internship at Oracle Labs

SC '15, November 15-20, 2015, Austin, TX, USA
© 2015 ACM. ISBN 978-1-4503-3723-6/15/11…$15.00
DOI: http://dx.doi.org/10.1145/2807591.2807620

1. The size of the graph is very large – larger than the capacity of a single machine's main memory.
2. Many small-sized random memory accesses over the whole graph instance are performed, especially during neighborhood iterations and graph traversals.
3. The computation-to-communication ratio is low – i.e., the amount of computation is small compared to the amount of data movement.
4. There is a large degree of inherent parallelism.

Note that the above characteristics (except the last one) are very different from classic computation-oriented workloads and thus cause performance issues with conventional systems. Specifically, the performance is governed by tremendous amounts of random data accesses. Moreover, the massive size of the data makes it hard to do the analysis within the memory space of a single machine.

Consequently, several different frameworks have been proposed that perform graph analysis on a cluster of machines. Section 2 reviews some of these frameworks including GraphLab [21] and GraphX [11]. However, the performance of these systems is disappointing, since even when running on top of many machines, the execution time is still much slower than running the same analysis within a single machine equipped with enough main memory [28].

In this paper we present PGX.D, a new distributed graph processing system which is fast and scalable, and share the lessons from building such a system. Section 3 describes (1) how our system implements low overhead, bandwidth-efficient communication with Run-to-Complete (RTC) cooperative threading and (2) how it keeps workloads well balanced not only among multiple machines, but also among multiple cores in each machine by applying edge partitioning and edge chunking. Section 4 shows that PGX.D's programming model is still intuitive but more flexible than conventional systems as it allows *data pulling*.

In section 5, we evaluate the performance of our system against two conventional distributed frameworks (namely GraphLab and GraphX) as well as against single machine optimized implementations. The experimental results reveal that PGX.D is not only by far faster than conventional systems but also fast enough to catch up with single machine optimized implementations. We also analyze the impact of our design choices on overall performance and share our lessons.

Our contributions can be summarized as follows:

- **A Very Fast Distributed Graph Processing System:** We present PGX.D, a fast *distributed* graph processing system which consistently outperforms existing systems to significant degrees (e.g. 3x - 90x faster than GraphLab). Moreover, PGX.D outperforms single-machine optimized implementations with 2 to 16 machines for all but one algorithm in our suite.

- **Low-Overhead Communication Mechanism:** We show that with an elaborately implemented cooperative context switching mechanism, one can exploit the bandwidth of the interconnection network with a reasonably low overhead. Such a low overhead is essential for fast execution of graph algorithms that consist of a lot of small-sized iteration steps.
- **Enabling Data Pulling:** Moreover, our communication mechanism enables us to apply the *data pulling* pattern directly in a cluster environment; this gives an additional performance improvement for computation-heavy algorithms by avoiding extra synchronization between multiple cores in each machine.
- **Reducing Traffic and Balancing Workloads:** We show that by replicating only a small number of high-degree nodes, one can reduce the amount of communication significantly. We also show that balancing the workload only among multiple machines is not sufficient. Instead it is necessary to balance the workload among multiple cores in each machine as well. We achieve such a workload balance by applying edge partitioning and edge chunking techniques together.
- **Justification for Beefy Clusters:** For large-scale graph analysis we advocate the use of *beefy* clusters [20], where machines with a large number of cores and a decent amount of memory are connected through a fast network. We show that one needs many cores to extract random access bandwidth from local memory as well as a fast network for inter-machine communication. Lack of either would become the performance bottleneck.

2. BACKGROUND

There are several different systems proposed for distributed graph processing. In this section, we review some of the common ideas among those systems and discuss them. We also explain how their shortcomings are resolved in our system.

Neighborhood Iteration And Communication

Originating from Pegasus [18] and Pregel [22], it has been observed that many interesting graph algorithms can be decomposed into iterative executions of certain computation kernels that are specific to each algorithm. Moreover, such kernels even share a typical computation pattern in which every node in (a subset of) the graph iterates over all of its (incoming or outgoing) neighbors in parallel, receives values from the previous iteration step, and computes new values for the current iteration step.

Consequently, distributed graph processing systems adopt more or less similar programming models, which are generalizations of the aforementioned computation patterns. These programming models include the GIM-V model in Pegasus, the Gather-Accumulate-Scatter (GAS) model in GraphLab [21], and the vertex-centric model in Pregel, Giraph [1] and GraphX [11].

However, all these models only support the *data pushing* communication pattern. That is, each node can only write some value to its neighbors (by using a reduction), but cannot read values from them. Note that this is an artificial limitation from the underlying framework – *data pushing*, or remote writes can be easily implemented as message passing in distributed environments, but not *data pulling*, or remote reads. The issue is that typical graph algorithms are naturally designed to use 'reads' rather than 'writes'. Therefore, the programmer needs to reconstruct the algorithms to use 'writes' in order to run his algorithms on these frameworks.

[Our Improvements] In PGX.D we provide a simple programming model where the above neighborhood iteration pattern can be expressed naturally. We provide the same expressiveness as existing programming models but further allow the *data pulling* communication pattern (Section 4); thereby, our programming model is very convenient compared to existing ones. Our experiments in Section 5 show, that the *data pulling* mechanism even gives a significant performance boost for certain algorithms.

Implementing an Efficient Communication Mechanism

The above iterative computation model allows communication schemes which are bandwidth-efficient: Since the computation of the current step is only dependent on the values from the previous step, the communication between nodes can be delayed until the end of the current step. Therefore, the system can buffer up many small messages and create a large network packet out of them, which utilizes the network bandwidth much more efficiently.

Naturally, all the existing graph processing systems apply the above technique, while some systems rely on underlying general frameworks for doing it. For instance, Pegasus is built on top of Hadoop and GraphX on top of Spark; GraphLab, Pregel and Giraph implement their own communication mechanism.

Nevertheless, these frameworks often introduce certain performance overheads for achieving such bandwidth-efficient communication. Not only are there overheads for message (de-)marshalling, but there are also overheads for scheduling a lot of concurrent vertex-computing tasks, whose small messages are buffered up. These framework overheads can account for a significant portion of total execution time, especially for algorithms that consist of many iterations of a small-computation kernel.

[Our Improvements] PGX.D implements a very efficient low overhead mechanism for communication and task scheduling (Section 4). Section 5 reveals that PGX.D outperforms the existing systems because of both its lower framework overhead and its better communication management.

Reducing Traffic and Balancing Workloads

A few techniques have been proposed for reducing communication traffic for distributed graph analysis systems. A direct approach would be to partition the graph such that the number of crossing edges is minimized; however, this is a well known NP-hard problem. On the other hand, approaches like the selective ghost node technique [26] or moving computation instead of data [31] involve relatively little computational overheads but tend to be effective. Yet, applying these techniques may impose some programming overheads for users.

Regarding workload balance, Low et al. [21] also pointed out that the naive vertex partitioning (i.e., each partition has similar number of nodes) may result in severe workload imbalance between machines, since real-world graphs have high skewness in their degree distribution. Instead, they proposed the edge partitioning scheme – i.e., each partition has a similar number of edges.

[Our Improvements] PGX.D makes it easy for the user to apply the selective ghost node and moving computation technique. As for workload balance, PGX.D applies not only the edge partitioning scheme, but also a new edge chunking strategy which is used in the context of task scheduling and proves to be essential to balance workloads between cores in each machine.

3. PGX.D: DESIGN AND IMPLEMENTATION

3.1 Overview

PGX.D is a distributed in-memory graph processing engine where the large graph instance is distributed over the aggregated memory of multiple machines in a cluster.

The design of PGX.D favors *beefy* computing clusters where

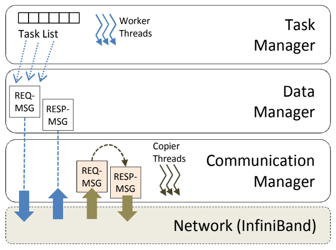


Figure 1: PGX.D Overview (one instance per machine)

each machine is loaded with fairly large amounts of memory as well as many CPU cores, and the machines are connected by modern high-bandwidth, low-latency networks. As we will see in Section 5, when the graph algorithm execution is dominated by small-sized random data movement, its performance in a distributed environment is bottlenecked by the network bandwidth *and* local DRAM bandwidth of each machine; it is essential for saturating either bandwidth to have a sufficient number of CPU cores working in parallel.

Just like other conventional distributed graph frameworks [21, 11, 22, 18], PGX.D supports the synchronous stepwise execution of graph algorithms. That is, the application execution is decomposed into multiple parallel execution steps, whereas the synchronization happens at the end of each step. Other sequential computations can also occur between parallel steps. As explained in the previous section, this execution model is well suited for typical graph algorithms which consist of a few parallel computational kernels that are repetitively invoked in iterative manners.

As for consistency model, however, PGX.D is different from conventional systems as it adopts a more relaxed model than the strict bulk-synchronous model. That is, (1) PGX.D allows one-sided reading of remote data (i.e. *data-pulling*) in a single iteration step and (2) PGX.D applies local and remote write requests immediately without waiting for the synchronization point at the end of the step. This relaxed consistency model allows performance benefits, but careless programming may cause non-deterministic results. See Section 4 for detailed discussions.

Figure 1 illustrates the block diagram of PGX.D as well as its basic execution flow. Note that the figure corresponds to one instance of the PGX.D process; the same program is instantiated on each machine in the cluster. As shown in the figure, PGX.D is composed of three layers: Task Manager, Data Manager and Communication Manager. Details of the implementation of each layer will be discussed in following subsections.

The figure also illustrates the basic execution flow:

(1) The graph algorithm is decomposed into numerous small execution contexts, i.e. tasks, that are stored in the Task Manager. Specifically, a set of *worker* threads is initialized by the Task Manager at system start up. Each worker grabs a chunk of tasks from the task list and executes them one by one.

(2) Whenever a task reads or writes graph data, it checks with the Data Manager which knows the location of the data. If the data resides in the current machine, the access is immediately resolved; otherwise, the Data Manager buffers up the request into a large request message. In case of such a remote read access, the task is blocked – the worker thread moves to the next task in this case.

(3) The blocked tasks are resumed when the response message for the corresponding request message comes back. This continuation mechanism will be discussed in the next subsection as well as in Section 4.

(4) Naturally, each PGX.D process is responsible for generating response messages for other processes in the cluster. This is done by the Communication Manager. Similar to the Task Manager, the Communication Manager pre-populates a set of *copier* threads at start-up time. Whenever a request message comes in, one of these copiers processes it and creates a response message if required.

3.2 Task Management

The Task Manager creates the list of tasks at the beginning of each parallel region. Once the task list is set and the parallel region begins, each worker thread grabs a chunk of tasks from the task queue and processes them sequentially by invoking their `run()` method. Note that the user program can create a different kind of task for each parallel region.

The task object in PGX.D is extremely light-weight. As a matter of fact, we do not store any architectural state (e.g. program counter) in this object. The invocation of the `run()` method completes no matter what. If the task has requested a remote data read inside the `run()` method, the execution will be continued by the same worker thread later, when the data becomes available.

Note that all the information which is needed after continuation should be explicitly stored in the private member fields of the task object or as temporary node properties. Section 4 discusses how the user can create tasks for his/her graph algorithms.

More interestingly, PGX.D does *not* keep a separate queue for the blocked tasks. Even when a task requests a remote read (or remote method invocation) inside, the `run()` method finishes normally. However, PGX.D guarantees continuation of this task when the data becomes available, with the following mechanism:

(1) A task (executed by a worker thread) can read (property) data from a different vertex by calling the `read_remote()` method. If the target vertex is indeed located in a different machine, the Data Manager buffers up this request in the request message. Note that request messages are accumulated separately by each worker.

(2) While buffering up the remote requests into a message, the Data Manager maintains a corresponding side data structure that logs the tasks the requests originated from, in the same order.

(3) When the message buffer has reached its maximum size or the worker thread has completed all tasks, the message is sent to the remote machine and the buffer is returned to a buffer pool. However, the corresponding side data structure stays in memory until the response message is received.

(4) When the response message is received from the remote machine, the message is delivered back to the worker thread which originated the read requests. Also the corresponding side data structure is retrieved as well. Using the side structure, the worker can iterate over the payload of the received message and invoke continuation methods on the corresponding task object (i.e. calling `read_done(void* buffer)`).

Note that since a task is always executed by the same single thread, there is no need to protect private fields of a task object with locks. This also makes tracking of unfinished tasks trivial. A particular job completes when the task list is empty and there are no unfinished remote requests.

3.3 Data Management

The Data Manager is responsible for maintaining graph data across multiple machines in the cluster. Internally, the graph (consisting of nodes N and edges E) is represented using the conventional Compressed Sparse Row (CSR) format, partitioned over multiple machines. Likewise, each node/edge property is represented as an $O(N)$/$O(E)$-sized array, partitioned over multiple machines.

We partition the vertices across the cluster: each node (with all its properties) in the graph is assigned to a machine. The partitioning happens while loading the graph into memory with the resulting partitioning information shared across all machines. More specifically, we assume that the vertices are numbered from 0 to $N-1$ by a preprocessing step. During loading, we decide how these vertices are partitioned into P machines. The partitioning strategy is discussed later in this section. However, each partition holds consecutive vertices from a numbering perspective, which allows us to identify each partition by its $P-1$ pivot node numbers. This information is shared by all the machines.

Once the graph is loaded and partitioned, we assign a global id to each node, a 64-bit number which concatenates the machine number and the local offset for that particular node. Using this representation, the Data Manager is able to quickly identify the location of a node. This also allows us to only transfer local offsets when sending remote addresses.

During loading time, the Data Manager applies the following techniques which are intended to reduce the communication traffic and improve the workload balancing (Section 2):

Edge Partitioning

Edge Partitioning [21] is a graph partitioning strategy that assigns a similar number of edges to every machine instead of a similar number of nodes. Note that the goal of this partitioning is *not* to reduce the communication (i.e. avoiding crossing edges), but to balance workloads between machines; for neighborhood iterating algorithms the amount of work is roughly proportional to the number of edges. PGX.D implements the edge partitioning technique at graph loading time. Before partitioning the graph, it first computes the total sum of in-degrees and out-degrees for all vertices. It then chooses the pivot vertices that result in a balanced sum of in-degrees and out-degrees for each partition.

Selective Ghost Node

Selective ghost node creation is a technique to choose a set of high-degree vertices and to duplicate *ghost copies* of them on each machine [26]. Consequently, each ghost node only keeps local edges that do not cross machine boundaries. In PGX.D, ghost nodes are created at loading time – PGX.D computes the in-degree and out-degree of each node and creates a ghost if either degree is larger than the specified threshold value.

During runtime, ghost nodes are synchronized to the original node at the boundaries of a parallel region, semi-automatically. That is, the synchronization mechanism is handled by the system as long as the program provides all the required information. Specifically, for each parallel region, the program needs to define what properties are used in the region as well as how they are used – to be read or to be written (reduced).

Once this information is available, ghost node synchronization is performed in the following way: for properties that are to be read in the parallel region, PGX.D copies the original values into the ghost nodes prior to the execution step. For the properties that are to be written (reduced), the *bottom* value is set to each ghost copy at the beginning – e.g. `0` for additive reduction. The partial results from ghost nodes are reduced to the original value at the end of the step.

Edge Chunking and Ghost Privatization

More importantly, we noticed that the same principles for edge partitioning and ghost node techniques can be used to solve intra-machine performance issues. That is, the key idea of edge partitioning can be used to solve the workload balance problem between multiple cores in the machine. Similarly, we can privatize ghost nodes to reduce the amount of atomic instructions in each machine. Specifically, we applied the following techniques:

1. **Edge Chunking:** Note that tasks are grouped into chunks, which in return are allocated to worker threads (Section 3.2). The Task Manager creates chunks by edge count, thereby ensuring that that each chunk will contain a similar number of edges instead of similar number of nodes. Consequently, workloads between cores are improved, since no worker thread would iterate much more neighbors than others.
2. **Ghost Privatization:** PGX.D creates *thread-private* copies of ghost nodes. Ghost privatization is, however, only applied if some properties are reduced in the parallel region; during the parallel region, reductions to the properties are applied to the thread-private copies without using atomic instructions. The ghost node synchronization becomes two-staged – first between cores and then between machines.

3.4 Communication Management

In PGX.D, the Communication Manager is responsible for (1) overall interaction between the underlying communication handler and for (2) creating response messages to remote machines. PGX.D maintains a dedicated thread for traffic control, namely the poller thread. As the name indicates, this thread polls across various queues between each workers and copiers and puts/gets message buffers to/from the networking device driver.

The Communication Manager layer is carefully designed to handle communication efficiently without introducing big overheads (or latency). Fast, low-overhead implementations were used for queues and buffer pools, while back-pressure mechanisms were induced to avoid deadlocks.

In addition, the Communication Manager controls the copier threads which process incoming request messages. As for write (reduction) requests, the copier applies them directly with atomic instructions. As for read requests, the copier creates a corresponding response message and sends it back to the originating machine.

The remote method invocation (RMI) is also handled by the copier threads, in a similar way than remote reads. At setup time, the PGX.D application registers its RMI methods and gets unique identifiers. At runtime, RMI request messages are encoded with this identifier, out of which the copier executes the appropriate method and generates response messages.

4. PROGRAMMING MODEL

4.1 Implementing User Algorithms

4.1.1 Neighborhood Iteration Tasks

Just like existing distributed graph processing systems, PGX.D provides an intuitive programming model for writing neighborhood iterating graph algorithms (see Section 2). For instance, consider the following example algorithm, which is a neighborhood iteration algorithm that uses the *data pushing* communication pattern.

```
foreach(n: G.nodes)  // for each node n
  foreach(t: n.outNbrs) // for its outgoing neighbors
    t.foo += n.bar;  // write n.bar into t.foo
```

To encode the above algorithm in PGX.D, the user needs to implement a neighborhood iteration task as follows:

```
class my_task_push : public outnbr_iter_task {
public:
   void run(..) {
   int bar_val = get_local(get_node_id(), bar);
   write_remote<SUM>(get_nbr_id(), foo, bar_val);
} }
```

The above is, in fact, the context object explained in Section 3 where the above `run()` method is executed for every edge in the graph in parallel [1]. The above code shows two API methods for accessing properties of nodes – `get_local()` for reading a local property and `write_remote()` for writing a remote property with a reduction operation (`SUM`). The remote write method, as explained in Section 3, is translated into a write request (and buffered) if the destination indeed resides in another machine, or processed immediately otherwise.

PGX.D supports the *data pulling* communication pattern as well. Consider the following case:

```
foreach(n: G.nodes)    // for every node n
  foreach(t: n.inNbrs) // from its incoming neighbors
    n.foo += t.bar       // read 'bar' and sum into n
```

Note that the existing graph processing systems do not allow this pattern, unless the programmer explicitly changes it into *data pushing* by flipping edge directions. To the contrary, with PGX.D, the user can directly encode the *data pulling* pattern as follows:

```
class my_task_pull : public innbr_iter_task {
 public:
  void run(..) { ..
    read_remote( get_nbr_id(), bar);
  }
  void read_done(void* buffer,..) { ..
   int val= get_data<int> (buffer);
   int curr= get_local(get_node_id(), foo);
   set_local(get_node_id(), curr+val, foo);
} }
```

Again, above is the context object whose `run()` method is invoked for every (incoming) edge in the graph. Inside this method, the current node is invoking the `read_remote` method for its incoming neighbors. If the other node is indeed located in a different machine, this method creates a remote read request and saves this context into the side structure (Section 3). When the response message is received, `read_done()` is invoked with a pointer to the fetched remote data, so that the execution is continued. If the other node is in the same machine, `read_done()` is immediately invoked with the pointer to the local data.

Using multiple worker threads, PGX.D processes several of above iterator contexts in parallel. However, it is guaranteed that all the (incoming) edges to the same (current) node are handled by the same worker thread, which eliminates any synchronization for updating the value `foo`.

PGX.D provides a general vertex deactivation mechanism using filters. Specifically, the user can implement a custom filter method which is evaluated for each vertex prior to its execution. The method returns true only if the current vertex is active. By using a boolean node property, the user can deactivate nodes from within the `run()` and `read_done()` method if a certain condition is met. The implementation for the respective filter would be very easy in this case:

```
class filtered_task : public outnbr_iter_task {
 public:
  bool filter() {
    return get_local(get_node_id(), active);
  }
  void run(..) {
    ..
    if(/* some condition */)
        set_local(get_node_id(), false, active);
  }
}
```

4.1.2 *Run-To-Completion Task Mechanism*

Fundamentally, the PGX.D programming model is built upon a more general, continuation-based context switching mechanism.

[1]the C++ code here is simplified a little for the sake of explanation in the limited space. For instance, type-related template parameters are omitted here.

```
void pgx_app::main(...) {
   ... // initialization
  // job creation
  job my_job1 = new job();
  List write_props = [(FOO1, ADD)];
  List read_props = [FOO2];
  my_job1->add_edge_iterator<task1>
                     (read_props, write_props);
  job my_job2 = ..

   // main execution
   while ( ... ) {
     // sequential computation
     ...
     // parallel region
    Task_Manager.begin_job(my_job1);
     // sequential computation
     ...
     // parallel region
    Task_Manager.begin_job(my_job2);
}  }
```

Figure 2: Example of PGX.D Application

For performance reasons, however, our implementation does not adopt any existing light-weight threading or fiber libraries that allow continuation from any random program location. Instead, we adopt the notion of run-to-completion (RTC) task management [7] as it imposes only small performance overhead.

In the RTC task model, the context switching does not happen at any arbitrary place in the task execution. Rather, the task must voluntarily yield the execution by returning from the entry method. It is up to the task to save all necessary context information so that execution can be continued later. Specifically, the task, or the user context in PGX.D, should implement the following interface:

```
class task {
  void run(...)=0          // entry of task
  void read_done(...)=0   // continuation from read
}
```

The user encodes the execution behavior of the task by implementing the `run` and `read_done` method. The `run` method is called to enter the task and invoke local/remote reads/writes. The `read_done()` method is the continuation point from the remote read (or from the remote procedure call as in Section 3.2). If the task needs to continue/discontinue more than once in an iteration, the user can implement a state machine to distinguish multiple callbacks to the `read_done()` method. PGX.D's scheduler ensures that the task callbacks are always executed single-threaded.

PGX.D tasks are defined with an iterator class which indicates a scheduling mechanism for the tasks. Specifically, PGX.D provides two iterators for implementing neighborhood iterating algorithms: the node iterator and the edge iterator (with incoming and outgoing variants). These built-in iterators let the PGX.D scheduler handle each common neighborhood iteration pattern with a different optimization. It is also possible for the advanced user to define his/her own iterator class so that PGX.D would schedule the tasks in a different manner than the built-in iterators.

4.2 Top-Level Execution Model

Figure 2 shows the top-level view of a PGX.D application where the application is decomposed into sequential and parallel computation regions. In PGX.D, a parallel computation region is represented as a job which in turn contains the actual parallel iteration kernel, or the task (Section 4.1). When handing the job to the Task Manger, PGX.D starts the parallel execution. To the contrary, between theses parallel jobs, the main algorithm is executed in a sequential manner.

In PGX.D, node and edge properties are represented in column-oriented ways. Consequently, each property can be referenced as a

Category	Item	Detail
SW	OS	Linux 2.6.32 (OEL 6.5)
	compiler	gcc 4.9.0
CPU	Type	Intel Xeon E5-2660
	Frequency	2.20 GHz
	Parallelism	2 socket * 8 core * 2 HT
DRAM	Type	DDR3-1600
	Size	256 GB
Size	# Machines	32
Network	Card	Mellanox Connect-IB
	Switch	Mellanox SX6512
	Raw BW	56Gb/s (per port)

Table 1: Experiment Environment

separate entity, and it is trivial to create or delete temporary properties. The user needs to specify the list of properties that are read and written for each job; reduction operators also need to be specified for the properties that are written. Then, PGX.D automatically takes care of synchronization of properties between ghost nodes between each job.

Note that the consistency model in PGX.D is more relaxed than the strict bulk-synchronous model in Pregel or MapReduce. In PGX.D the updates in one iteration step are applied directly without waiting for the beginning of the next iteration step. Therefore, PGX.D does not guarantee deterministic behavior on properties that are both written/reduced and read within the same parallel region. The user needs to avoid this situation by creating temporary copies of properties, which is trivial in PGX.D. Note that PGX.D's consistency model is, however, a natural one; conventional in-memory parallel frameworks such as OpenMP use the same model.

4.3 Compiler Support

PGX.D provides a fairly straightforward programming model for neighborhood iterating graph algorithms (Section 4.1). Still, for the sake of data scientists who may not be experts in C++ programming, we added support for Green-Marl [15] – a Domain-Specific Language (DSL) for graph algorithms. Note that all the graph algorithm pseudo-codes in this paper are, in fact, written in Green-Marl syntax.

Specifically, we extend the Green-Marl compiler such that it can transform a neighborhood iterating algorithm like Pagerank into an equivalent PGX.D application. The detailed design and implementation of the compiler is clearly outside the scope of this paper. However, the basic techniques are quite similar to a previous publication [16] where the authors compiled Green-Marl programs for a Pregel-like system.

To be clear, all the experiments in the Section 5 are done with our hand-written codes, although the compiler-generated ones give almost the same performance. Our next goal is to extend the compiler so that it can even translate algorithms that are not neighborhood iterating into PGX.D using our general task framework.

5. EVALUATION AND DISCUSSION

5.1 Methodology

All the experiments in this section are conducted in our own cluster, which is composed of identical machines. Table 1 summarizes the hardware, software and network configuration of our cluster. Note that our cluster uses InfiniBand, a high-end commodity interconnection fabric, as its backbone network. However, the implementation of PGX.D is independent from the underlying interconnection fabric, as it does not exploit any of its special features (e.g. RDMA).

Name	Description	GL	GX
PageRank: Exact	Exact PageRank computation	-	-
PageRank: Approx	Approximate PageRank with delta propagation	v	v
WCC	Weakly Connected Components	v	v
SSSP	Single source shortest path (Bellman-Ford)	v	v
Hop Dist	Breadth-first traversal from the root	v	-
EigenVector(EV)	Compute eigenvector centrality (first component)	-	-
KCORE	Find Biggest K-core number	v	-

Table 2: Algorithms used in the experiments: the third and fourth column indicates whether builtin algorithms were available in the GraphLab(GL) and GraphX (GX) package.

5.2 Performance Comparison

We compare the performance of our system against state-of-the-art *distributed* graph processing engines, namely GraphLab and GraphX. We do this comparison by running the same algorithms on the same graph instances using the same cluster hardware but with different frameworks. In this experiment, we used GraphLab version 2.1 and Spark/GraphX version 1.1.0, both of which are publicly available. For GraphLab executions, we use the synchronous version of their computing engine as it performed consistently faster than the asynchronous one. For GraphX executions, we tried different partition strategies and numbers of partitions and chose the one with the best results. For all systems, we made sure that every core in the machine is actually utilized and the communication is done through the InfiniBand fabric.

For our experiments, we used popular graph algorithms that are common in the literature [28, 23, 21] for evaluating performance of graph processing systems. The list of these algorithms is summarized in Table 2. For GraphX and GraphLab execution, we used the built-in implementation of the algorithm if it was available in the package; we implemented the algorithm on top of these systems by ourselves, otherwise. In addition, we implemented these algorithms as standalone applictions using direct CSR (Compressed Sparse Row) arrays and OpenMP parallel loops [15]. And we measured the performance of these applications as single machine performance that do not suffer from any framework overhead.

Among these algorithms, let us consider *Pagerank* as a typical example. Pagerank can be computed with the power iteration method where the following kernel is repeatedly computed until converged:

```
foreach(n: G.nodes)     // for every node
  foreach(t: n.inNbrs) // read data from its in-nbrs
    n.PR_nxt += t.PR / t.degree();
```

Although the above method is the preferred way of computing Pagerank for single machine environments, it is not the case for distributed environments. This is because the above method requires *pulling of data* from remote nodes, which is expensive or even disallowed in distributed frameworks. Instead, the following variant is preferred in distributed execution:

```
foreach(n: G.nodes)     // for every node
  foreach(t: n.outNbrs) // write data to its out-nbrs
    t.PR_nxt += n.PR / n.degree();
```

Note that the above variant writes data to outgoing neighbors, instead of reading data from incoming neighbors. Fundamentally, it performs the exact same computation as before, but is more friendly for distributed frameworks because it only requires *pushing of data* to remote nodes. In this experiment, we implemented the *push* version of Pagerank for GraphLab and GraphX, while we implemented the *push* and the *pull* version for PGX.D, since only PGX.D allows both directions of data movement.

Both GraphLab and GraphX are packaged with another variant of Pagerank, which only computes approximate values for the sake

		PR (pull) (per iter)		PR (push) (per iter)		PR (approx) (per iter)		WCC (total)		SSSP (total)		HopDist (total)		EV (per iter)		KCore (total)			
		TWT	WEB	TWT	WEB	TWT	WEB	TWT	WEB	TWT	WEB	TWT	WEB	TWT	WEB	LJ	WIK	TWT	WEB
SA	1	1.92	0.45	3.29	11.0	0.71	0.83	8.70	3.54	18.8	35.1	2.44	2.81	1.20	0.38	5.62	21.5	194	1077
GX	2	-	-	305	n/a	19.2	17.6	638	1380	n/a	n/a	1140	4925	1286	2117	n/a	n/a	n/a	n/a
	4	-	-	107	1320	59.0	34.9	343	741	1237	n/a	467	2634	511	834	n/a	n/a	n/a	n/a
	8	-	-	66.0	155	37.5	26.2	210	540	811	6167	386	1824	284	679	n/a	n/a	n/a	n/a
	16	-	-	30.3	79.1	32.4	18.6	207	646	645	4570	263	1257	119	792	n/a	n/a	n/a	n/a
	32	-	-	32.6	62.1	24.3	20.0	280	806	601	4582	307	1440	60.9	575	n/a	n/a	n/a	n/a
GL	2	-	-	15.1	25.1	5.64	5.52	353	638	101	198	11.1	36.2	28.3	45.5	2664	10365	n/a	n/a
	4	-	-	11.9	21.0	2.92	3.72	179	207	62.5	153	8.30	34.0	25.9	48.5	1648	6794	n/a	n/a
	8	-	-	9.06	16.3	2.87	3.55	93.2	193	43.4	104	8.00	30.2	18.8	33.4	2106	4770	n/a	n/a
	16	-	-	6.21	11.6	2.16	2.40	59.3	85.3	35.0	82.1	7.00	30.2	12.8	24.3	1440	5459	n/a	n/a
	32	-	-	5.96	7.25	2.49	1.83	33.6	45.0	37.2	93.6	6.20	24.7	8.85	10.9	1215	3878	n/a	n/a
PGX	2	4.14	1.78	4.57	2.72	1.00	0.31	11.5	14.5	27.2	50.5	4.43	7.82	2.95	1.50	91.8	197	2270	5145
	4	2.40	1.15	3.73	1.46	0.76	0.18	7.78	7.83	19.1	23.8	3.31	3.67	1.69	0.78	84.4	99.1	1502	3531
	8	1.28	0.62	2.81	0.88	0.54	0.13	5.20	4.81	14.6	13.5	2.22	2.55	0.90	0.45	70.2	99.0	871	2643
	16	0.60	0.39	1.55	0.59	0.41	0.087	2.68	2.68	8.17	9.58	1.26	1.97	0.50	0.27	60.9	106	649	1946
	32	0.36	0.27	0.88	0.35	0.25	0.066	1.74	1.61	5.07	6.32	0.81	1.95	0.34	0.19	54.7	115	579	1173

Table 3: Execution Time of Algorithms on each system with different number of machines: execution time is measured in seconds. n/a stands for the case when the algorithm was not finished within three hours (including loading) or when the system fails to execute algorithm.

			Loading Time		
Name	# Nodes	# Edges	GX	GL	PGX
LiveJournal(LJ) [5]	4,847,571	68,993,773	7.42	88.3	4.23
Wikipedia(WIK) [2]	15,172,740	130,166,252	8.67	171	19.5
Twitter(TWT) [19]	41,652,230	1,468,365,182	41.9	638	92.5
Web-UK(WEB) [6]	77,741,046	2,965,197,340	75.5	3424	76.6

Table 4: Name and size of the graphs used in the experiment: the last three columns shows loading time of each graph into three systems, GraphX(GX), GraphLab(GL), and PGX.D. Loading times are measured in seconds; they include reading the data file and setting up the distributed data structure for each system. Note that each system uses a different file format.

of fast execution. This algorithm is implemented as follows:

```
// consider only active nodes
foreach(n:G.nodes)(n.active)
  foreach(t:n.outNbrs)
    t.delta_nxt += n.delta / n.degree();
// deactivate node, if delta is small
foreach(n:G.nodes)(n.active)
  if(n.delta_nxt < threshold)
    n.active = false;
// repeat until all nodes are deactived
```

Essentially, the approximate algorithm computes the difference (i.e. *delta*) of Pagerank values across iterations. Moreover, if the *delta* value of a node is smaller than the given threshold, the node is considered to be converged and becomes *deactivated*; at each iteration, only *active* nodes are considered in computation. Therefore, this method performs a decreasing amount of computation and communication as the iteration continues, since more and more nodes get deactivated. Also note that this approximation only works with the *push-based* implementation.

Other algorithms show similar computation patterns to either variant of Pagerank. WCC, SSSP, and HopDist are like approximated Pagerank – at each iteration step, only an active subset of nodes push their computed values to their neighbors. In WCC, a deactivated node can later be active again, however. On the other hand, EV is similar to exact Pagerank computation – every vertex is computing a new value from its neighbors at every iteration step. PGX.D implements this algorithm with data pulling.

We ran all the algorithms for each system on two large real-world graph instances – the follower relationship between Twitter users (TWT) [19] and link relation of webpages in uk domain (WEB) [6]. The only exception is the KCore algorithm, whose execution time on these graphs are prohibitively large for the GraphX and GraphLab framework. For this case, we used two other public graph instances with smaller size instead. Table 4 summarizes all graph instances used in the experiment. The SSSP algorithm uses edge weights. We generated these values using a uniform random distribution.

The results of the experiments are shown in Figure 3 as well as in Table 3. Figure 3 plots the relative performance among the three systems on two graph instances, while varying the number of machines. Specificially, the performance is normalized by GraphLab's execution time with two machines – i.e. a relative performance of 3.0 indicates that the system is three times faster than GraphLab's execution with two machines.

Table 3 provides the actual execution time measured in seconds. For Pagerank (exact and approximate) and Eigenvector, we report (average) per-iteration execution time, because one can choose arbitrary number of iterations for these algorithms. For other algorithms, we report the total execution time. All the numbers are the average of multiple runs, although there was not much time difference between them.

The plots in Figure 3 show that PGX.D outperforms GraphLab and GraphX significantly across all cases. Even though GraphLab is still an order of magnitude faster than GraphX, PGX.D is faster than GraphX by another order of magnitude in most cases. Moreover, PGX.D shows better scalability than the other two systems in a sense that the performance difference typically becomes larger for experiments running on many machines. GraphX showed the worst scalability of the three systems.

Note that the pull-based Pagerank implementation is significantly faster than the pushed-based version. This is because even though both implementations require the same amount of communication, they require different synchronization schemes between threads for the parallel computation. The push-based method uses *atomic addition* operations, because multiple threads may accumulate different remote values to the same node at one time. To the contrary, the pull-based method is implemented with a normal addition operation, since one node is updated by one thread only.

The dotted horizontal red and yellow lines in the plots indicate the relative performance of single-machine in-memory standalone execution (i.e. SA in Table 3) for the given algorithm on the given graph instance. Notice that the execution times of GraphLab and GraphX, even with 32 machines, are still a lot slower than the standalone execution of the same algorithms.

This is the combined result of various performance overheads from these frameworks and the inevitable communication overhead, as observed by Satish et al [28].

To the contrary, PGX.D is fast enough to outperform the single-machine execution, with 4 to 16 machines for most algorithms.

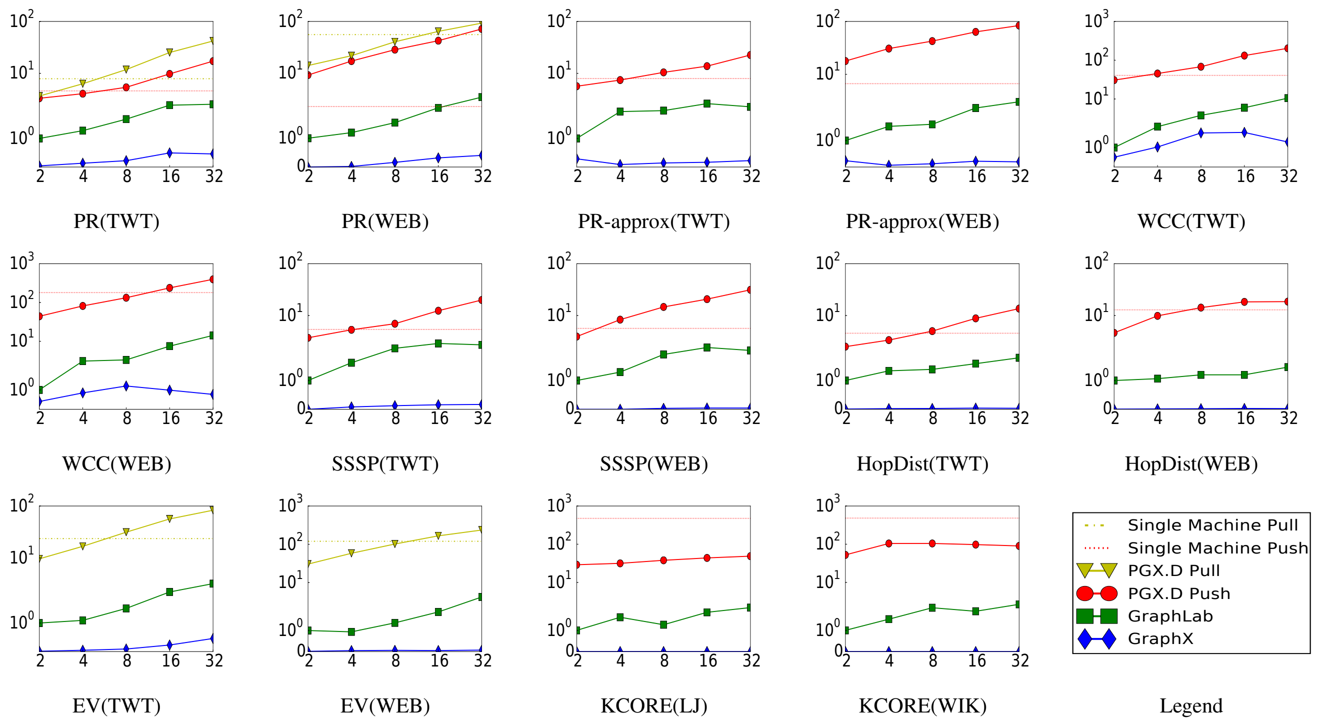


Figure 3: Performance Comparison: The X-axis shows the number of machines. The Y-axis denotes the relative performance with GraphLab's execution time on two machines as baseline. The dotted horizontal lines indicate the relative performance of single-machine in-memory standalone execution.

The only exception is the KCore algorithm, which requires a very large number of iteration steps. In such a case, even the relatively small overhead of PGX.D became significant when accumulated over iterations; still, PGX.D's performance gets closer to performance of a single machine with larger graphs (Table 3). Moreover, even larger overheads of GraphX and GraphLab prohibited us from running the KCore algorithm on large graph instances in time at all.

The numbers in Table 3 only account for the actual computation time but omit the time for loading data into the systems. This is for fair comparison between these systems, as they are reading the same data but from different formats – PGX loads from a binary file format while GraphX and GraphLab load from a text file. Yet, the data loading time for each system can be found in Table 4.

Note that the absolute execution time allows the reader to compare various systems from different reports, although only in a rough and indirect way. For instance, Fan et al. reported 8000 secs for computing WCC on the WEB graph instance, in addition to 2000 secs for building an index structure, with a RDBMS Engine (SQLServer) running on a machine that is comparable to ours [10].

5.3 Performance Analysis

In the previous subsection, PGX.D showed impressive performance advantages over existing graph processing systems. Although the various design decisions in PGX.D affect the overall performance, their contributions can be roughly grouped into the following two categories:

- **More Efficient Communication**: PGX.D is designed to handle a large degree of small-sized communication between multiple machines in an efficient manner. Especially, PGX.D exploits bandwidths of underlying network fabric without the framework introducing excessive overhead.
- **Better Workload Balance**: PGX.D adopts several techniques in order to achieve better workload balance among multiple machines and multiple cores – selective ghost nodes, edge partitioning, edge chunking, and ghost privatization.

5.3.1 Communication and Framework Overhead

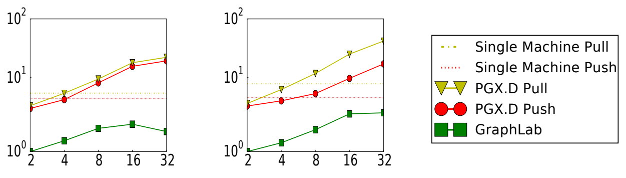
Efficient Communication

In order to separate out the impact of workload balance and efficient communication, we conducted the following additional experiment. First, we created a uniform random (i.e. Erdős-Rényi) graph, that is similar in size with the TWT graph: 40M nodes and 1.4B edges. Then, we measured the performance of PGX.D and GraphLab running Pagerank (Exact) algorithm on this instance. Note that due to the nature of the graph, no matter how partitioned, $(P-1)/P$ of the edges would remain as crossing edges for every partition. On the other hand, the workload between multiple machines and cores would remain balanced. Finally, every node is communicating with all of its neighbors at each iteration; thus communication is the biggest factor in this case.

Figure 4 plots the result of this experiment, where UNI indicates the case of uniform random graph. The plot also includes the case of TWT graph, for the sake of comparison. In the figure, PGX.D still outperforms GraphLab significantly for the uniform random graph, which confirms the efficiency of PGX.D's communication mechanism. The performance difference becomes even larger for TWT graph's case, especially for Pull-based implementation that is free of conflicting atomic writes. This is the effect of the well balanced workload between machines and cores of PGX.D.

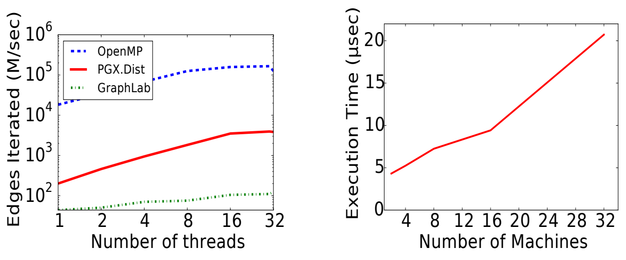
Framework Overhead and Barrier Latency

PGX.D imposes only small overheads for the framework execution itself, at least much less than GraphLab. To show this, we use a simple algorithm that iterates over all the edges in the graph without doing actual communication at all. We implement this algo-



PR(UNI) PR(TWT) Legend

Figure 4: Performance Comparison of PageRank (Exact) Algorithm on Uniform-Random Graph: Y-axis is the relative performance as in Figure 3. TWT graph is also included for comparison's sake.



(a) (b)

Figure 5: (a) is the measurement of edge iteration speed (millions per second). Performance was measured from time for iterating all the edges in the twitter graph with a single machine. (b) is the barrier execution time for PGX.D

rithm on top of PGX.D, GraphLab and as a standalone application with OpenMP. Then, we measure the time for iterating all the 1.4B edges of the Twitter graph using a single machine, while varying the number of (worker) threads.

Figure 5.(a) illustrates the result of this experiment, where each line stands for the edge traversal speed for each system. The OpenMP version achieves the fastest iteration, since it only performs a simple `for` iteration over CSR arrays. However, PGX.D still achieves impressive performance and is significantly faster than GraphLab.

Also, the latency of the *barrier* operation can affect the overall execution time for iterative graph algorithms, as they do one barrier operation at every iteration step. Figure 5.(b) measures the latency of the barrier implementation in PGX.D, varying the number of machines. Note that the time for the barrier operation is very small compared to the execution time of algorithms in Table 3.

In general, the performance impact of framework overhead and barrier latency varies for the type of algorithm. For algorithms which require a lot of computation and communication at every iteration (e.g. Pagerank), their impact is much less significant than an efficient communication scheme or keeping workload balanced. However, for algorithms which require a lot of iteration steps while each step does a very small amount of work (e.g. KCore), the performance is totally governed by these overheads. Most algorithms are somewhere in between of these two extremes.

5.3.2 Balancing Workloads and Reducing Traffic

Impact of Ghost Nodes

We evaluated the effect of ghost nodes with the following experiment. We set different threshold values for ghost node creation and measured both the reduction in utilized network bandwidth and the runtime of our algorithm. Specifically, we used the 8 machine execution of the Pagerank-Pull algorithm on the TWT graph instance which exhibits high skew in its degree distribution.

The result of this experiment can be found in Figure 6.(a) which shows that ghost nodes have a big impact on reducing communication traffic. By increasing the number of ghost nodes, we can reduce the traffic by about 50%. The runtime, however, is only reduced to 75% if we use about 500 ghost nodes and does not decrease further even if we add more. In other words, saving on traffic only results in a speedup up to a point until the network no longer becomes the bottleneck.

Impact of Edge Partitioning and Edge Chunking

Similarly, we measure the performance impact of the edge partitioning technique with another experiment. In this experiment, we again measure the performance of the Pagerank-Pull algorithm on the TWT graph instance with varying number of machines. However, we compare the performance when the edge partitioning is applied against when the vertex partitioning is applied. Vertex partitioning is the naive way of partitioning where each machine gets a roughly equal amount of vertices. Note that the ghost node technique is applied for both partioning schemes.

The result is summarized in Figure 6.(b) which clearly shows that the edge partitioning leads to much better performance in general. Naturally, the performance benefit grows larger as the number of machines increases, since the workload imbalance from vertex partitioning becomes worse.

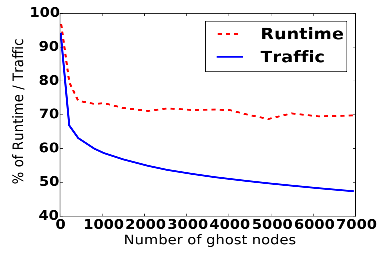
Figure 6.(c) provides a closer look at the effect of our load balancing techniques: edge partitioning and edge chunking (Section 3.3). The figure compares three configurations: the first column is the case where we only apply the ghost node technique, but use vertex partitioning and node-based task chunking. The second column shows the effect of applying edge partitioning on top of the baseline. Finally, in the third column, we show the result of edge chunking in addition to edge partitioning.

We observed that when using machines with many cores like ours, simple edge partitioning only gives little performance gains if applied without edge chunking together. This is because cores that handle a large number of edges become the bottleneck of the whole execution. The benefits of edge partitioning and edge chunking depend on the particular graph instance and have more impact for skewed graphs. Furthermore, the benefits depend on how many nodes are active during a computation. For the worst case execution (every vertex is active), edge partitioning balances the workload evenly. For the opposite case (i.e. almost every vertex is deactivated), the workload balance no longer becomes the bottleneck but instead the framework overhead dominates the execution time (see 5.3.1).

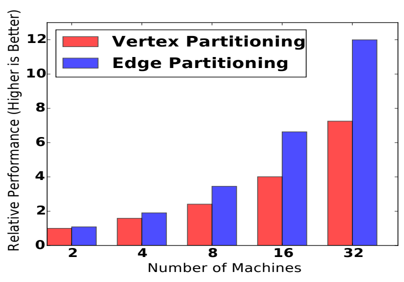
5.3.3 Worker/Copier thread exploration

PGX.D heavily uses the multi-core aspect of modern computer systems by populating multiple threads for workers and copiers (Section 3.1). Figure 7 explores the performance impact of having different numbers of threads for each type. The experiment was performed while running Pagerank Pull algorithm on the TWT graph with 16 machines.

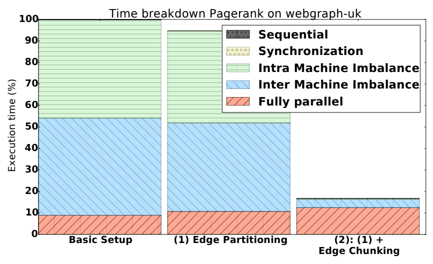
The figure shows a color coded representation for the measured relative performance among different worker/copier configurations. The best performance was archived with 16 to 20 workers and 8 to 16 copiers. Since our machines have 32 hardware threads, this shows that a small over-subscription can be helpful to increase performance, although not much. More importantly, Figure 8.(a) shows that the overall performance becomes bad if either kind of thread is not populated sufficiently. We used 16 workers and 8 copiers for all other experiments to have a fair comparison. How-



(a) Ghost Node Effect (PR on TWT)



(b) Edge Partitioning Effect (PR on TWT)



(c) Breakdown of Execution Time

Figure 6: (a) shows the effect of ghost nodes on performance and traffic for 4 machines. The Y-axis is the relative amount of runtime and traffic, compared to when no ghost nodes are used. (b) is the performance effect of edge partitioning. The Y-axis is the relative performance. (c) is the breakdown of performance for edge-based load balancing: Fully Parallel accounts for the time when all workers are busy. Inter Machine Imbalance is for the time when at least one machine is idle. Intra Machine Imbalance means when some workers are waiting for others in the same machine.

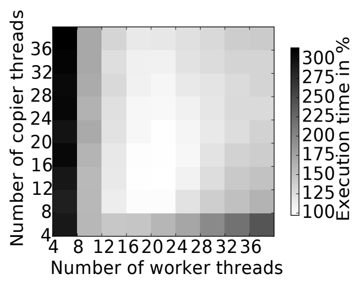
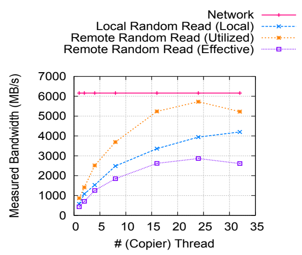
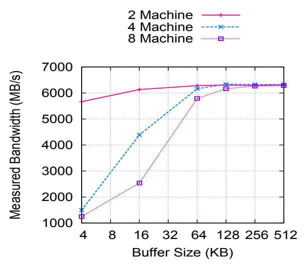


Figure 7: Exploration of the performance impact of using different number of worker/copier threads.



(a) Bandwidth Exploration



(b) Buffer Size Exploration

Figure 8: (a) shows the attainable remote random read bandwidth and network bandwidth. Note that DRAM bandwidth for random reads is naturally much smaller than for sequential reads. (b) shows the measured network bandwidth when changing buffer size for 2, 4 and 8 machines.

ever, the number of threads can be tuned manually for every algorithm in order to further increase PGX.D's performance. Eventually, the system will be able to auto-tune the number of threads based on the algorithmic workload.

5.3.4 Network bandwidth explorations

Remote random read bandwidth

We measured the remote random read bandwidth obtained with PGX.D. For this purpose, we wrote a micro-benchmark where a few threads continuously generated remote read requests. We used 8 byte addresses to get 8 bytes worth of data from a random remote address, and measured the obtained bandwidth while changing the number of copiers.

Figure 8.(a) presents the result of this microbenchmark when using two machines (i.e. 1:1 communication). There are four lines in the figure. The line labeled as 'Network' represents the available network bandwidth. There are two lines that are labeled as 'Remote Random Read': 'Utilized' bandwidth is computed by taking the total number of bytes transferred (address and data) divided by the elapsed time, while Effective' bandwidth only concerns data. Note that the 'Utilized' bandwidth is always twice the 'Effective' bandwidth, since both address and data are 8 bytes. Finally, for the sake of comparison, we added a line for 'Local' bandwidth – the bandwidth for 8-byte random reads on local DRAM with specified number of threads.

Figure 8.(a) reveals an interesting fact. First note that the 'Utilized' bandwidth is limited by the 'Network' bandwidth and the 'Effective' bandwidth is limited by the 'Local' bandwidth. The Figure shows that the remote access bandwidth is limited by the local DRAM bandwidth when only a small number of cores (for copier threads) is used. The network bandwidth becomes meaningful only after a sufficient number of cores is used. Therefore, the above observation reinforces our preference for using a *beefy* cluster for graph analysis. That is, an ideal cluster for large graph analysis would consist of machines with sufficient cores to extract as much random access bandwidth from the local DRAM as possible. Moreover, these machines need to be connected by an efficient interconnection network to balance the system.

Network buffer size

A proper size of the buffers used to send messages is crucial for exploiting the available network bandwidth. In order to determine which buffer size is optimal, we conducted the following measurement: We let each machine send dummy buffers to all other machines while varying the buffer size. We measured the attained bandwidth for 2, 4 and 8 machines. The result in Figure 8.(b) shows that large messages are essential for fully exploiting the available network bandwidth. For instance, when using 4KB message sizes, one can only make use of 1.5GB/s bandwidth out of the achieved maximum of 6.2 GB/s when performing N:N communication on 4 machines. The buffer size of our system has been set to 256KB based on this result.

6. RELATED WORK AND OUTLOOK

6.1 Existing Graph Processing Approaches

There are existing systems that focus on fast execution of graph analysis algorithms. First, there exist numerous graph libraries in many different programming languages. Examples include NetworkX [4] (Python) , Stinger [9], and Galois [25] (C++). Although these libraries provide low-overhead fast execution of graph algorithms, they do not support distributed execution and thus cannot handle graph instances larger than the single machine's memory.

Second, there are graph processing frameworks which focus on distributed execution of graph algorithms [21, 1, 22, 18, 29, 11, 24]. Pegasus [18] is an graph processig implementation built directly on top of a Map-Reduce system and thus suffers from disk overhead at each iteration. In contrast, Giraph [1] and Pregel [22] stay in-memory over iterations and perform message passing communication directly, even though they still make use of Map-Reduce schedulers. Grappa [24] is a runtime system for large-scale applications with some focus on graph analysis developed to run on commodity clusters. GraphLab [21] is a custom distributed framework that is generally regarded as one of the fastest distributed graph processing systems. GraphX [11] is similar to Giraph regarding that it implements graph processing features on top of a more general system, namely Spark. SociaLite [29] is a distributed Datalog based framework, optimized for analyses on social graphs.

In Section 2, we reviewed common ideas and techniques from these systems, and how PGX.D develops these ideas even further. Note that other studies [28, 23] confirmed that the performance of these existing systems are lower than the performance of single machine standalone implementations with enough memory. In this paper, we showed that PGX.D outperforms both distributed graph processing systems and single-machine implementations.

Third, there are so-called graph databases, or data management systems that directly support the graph data model. Neo4J [3] is a popular graph database that supports the new property graph model while Virtuoso [3] supports the classic RDF model. It has been noted that while these systems are designed for graph data management they are not necessarily ideal for graph analytics – although it is possible to implement graph algorithms on top of their API, their performance was again much slower than standalone single machine execution [23, 30].

Finally, there are approaches such as Fan et al.'s [10] that use conventional SQL engines for graph processing. Their rationale is that even though SQL engines might run slower than specialized graph engines, the difference is not huge, especially when considering the loading time. Moreover, by using SQL engines, users can exploit powerful SQL queries for browsing the analysis results – e.g., find the top-100 Pagerank nodes that have less than 1000 neighbors.

The exceedingly fast execution time of PGX.D weakens the above execution time argument. In addition, enterprise systems can typically endure data movement time since transactional systems and analysis systems (e.g. data warehouse) are often distinguished anyway. Nevertheless, PGX.D aims to implement a fast loader that can directly load/export data from/to database in fast manners. In addition, simple SQL operators can be implemented directly on top of PGX.D for the convenience of post processing.

6.2 Future Improvements

To improve both the usability and practicality of PGX.D, we plan to take several steps to extend the system. Furthermore, we want to share some technical challenges that we think should receive attention from the research community.

- **Building a long-running, interactive, multi-tenant system:** We are extending PGX.D as a long-running server system which allows multiple concurrent clients. That is, each client can load up multiple graph instances and execute different analysis algorithms on them in an interactive manner. Note that this opens up a set of interesting system design challenges. For instance, how should the system assign memory and CPU resources between clients while achieving overall fairness and efficiency? Also, the system should provide isolation and data consistency between clients without introducing too much memory overhead.

- **Adding more abstractions:** Fundamentally, PGX.D is an efficient, distributed computational system that provides abstractions for representing two-dimensional data and graphs. Our plan is to build additional abstraction layers on top of the PGX.D engine in order to support a wider range of applications. For instance, it would be relatively straightforward for us to provide abstractions for one dimensional data representations, which would suffice various non-graph workloads as in many existing Hadoop [13] or Spark [32] applications. An interesting challenge is, however, supporting higher dimensional data representations, or hypergraphs, as they are often used as a mathematical abstraction for machine learning and data mining algorithms. Currently, only a few distributed systems support hypergraphs directly [17].

- **Solving pattern matching queries:** The problem of subgraph isomorphism [14, 27, 33, 12] is about identifying all sub-graph instances in a large data graph that match the given (small) query graph. The problem is often referred to as *graph pattern matching* or *graph query*. Our plan is to extend PGX.D so that it can perform both computational analytics and pattern matching queries for large graphs in distributed environments. Fast distributed execution of graph queries, however, requires additional considerations on top of efficient communication and task-switching mechanisms. For instance, pattern matching algorithms tend to generate a potentially exponential number of partial solutions, or *match contexts*; careless implementation could result in either too much communication or too much memory consumption.

- **Support for dynamic graphs:** Real world graphs often do have a very large size and are also constantly changing over time (dynamic graphs). There are approaches to apply continuous pattern matching techniques to such graphs [8]. Since this field is an emerging area and can possibly solve many real world problems, our goal is to add support for continuous pattern matching techniques on dynamic graphs to PGX.D, while keeping its ability to perform classical computational analytics by using snapshots of these graphs for algorithms which do not support graph updates.

7. CONCLUSION

In this paper we presented PGX.D, a fast distributed graph processing system which assembles several state-of-the-art techniques. We showed that PGX.D outperforms existing systems significantly while it shows good scaling. In addition, PGX.D running with a small number of machines is faster than single-machine standalone executions. We pointed out that it is crucial for overall performance to have a low-overhead bandwidth-efficient communication mechanism with support for remote data pulling. We also showed techniques to reduce network traffic and to balance workloads between machines and evaluated their impact on performance. Furthermore we observed that it is necessary to balance workloads not only between machines but also between cores and we discussed the impact on performance when applying these techniques together.

8. REFERENCES

[1] Apache Giraph Project. http://giraph.apache.org.

[2] Koblenz Network Collection. http://konect.uni-koblenz.de.

[3] Neo4j graph database. http://www.neo4j.org/.

[4] NetworkX. https://networkx.github.io.

[5] SNAP. http://snap.stanford.edu/data/.

[6] Yahoo! Labs Datasets. http://webscope.sandbox.yahoo.com/.

[7] Atul Adya, Jon Howell, Marvin Theimer, William J Bolosky, and John R Douceur. Cooperative task management without manual stack management. In *USENIX Annual Technical Conference (ATEC)*, pages 289–302, 2002.

[8] Sutanay Choudhury, Lawrence Holder, George Chin, Khushbu Agarwal, and John Feo. A selectivity based approach to continuous pattern detection in streaming graphs. *arXiv preprint arXiv:1503.00849*, 2015.

[9] David Ediger, Robert McColl, Jason Riedy, and David A Bader. Stinger: High performance data structure for streaming graphs. In *High Performance Extreme Computing (HPEC)*, pages 1–5, 2012.

[10] Jing Fan, Adalbert Gerald Soosai Raj, and Jinnesh M. Patel. A case against specialized graph analytics engines. In *7th Biennial Conference on Innovative Data Systems Research (CIDR)*, 2015.

[11] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[12] Sairam Gurajada, Stephan Seufert, Iris Miliaraki, and Martin Theobald. Triad: a distributed shared-nothing rdf engine based on asynchronous message passing. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 289–300. ACM, 2014.

[13] Apache Hadoop. http://hadoop.apache.org/.

[14] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. Turbo iso: towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Proc. of the 2013 ACM SIGMOD International Conference on Management of Data*.

[15] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. Green-Marl: A DSL for Easy and Efficient Graph Analysis. In *ASPLOS*. ACM, 2012.

[16] Sungpack Hong, Semih Salihoglu, Jennifer Widom, and Kunle Olukotun. Simplifying scalable graph processing with a domain-specific language. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 208–218, 2014.

[17] Jin Huang, Rui Zhang, and Jeffrey Xu Yu. Technical report: Hyperx a framework for scalable hypergraph learning. 2015.

[18] U Kang, Charalampos E Tsourakakis, and Christos Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In *IEEE International Conference on Data Mining (ICDM)*, pages 229–238, 2009.

[19] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a social network or a news media? In *WWW '10: Proceedings of the 19th international conference on World wide web*, pages 591–600. ACM, 2010.

[20] Willis Lang, Stavros Harizopoulos, Jignesh M Patel, Mehul A Shah, and Dimitris Tsirogiannis. Towards energy-efficient database cluster design. *Proceedings of the VLDB Endowment*, 5(11):1684–1695, 2012.

[21] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[22] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A System for Large-scale Graph Processing. In *SIGMOD '10*, pages 135–146. ACM.

[23] Robert Campbell McColl, David Ediger, Jason Poovey, Dan Campbell, and David A Bader. A performance evaluation of open source graph databases. In *Proceedings of the first workshop on PPAA*. ACM, 2014.

[24] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Grappa: A latency-tolerant runtime for large-scale irregular applications. Technical report, Technical report, University of Washington, 2014. URL http://sampa. cs. washington. edu/papers/grappa-tr-2014-02. pdf. 4.1, 2014.

[25] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 456–471. ACM, 2013.

[26] Roger Pearce, Maya Gokhale, and Nancy M Amato. Faster parallel traversal of scale free graphs at extreme scale with vertex delegates. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 549–559, 2014.

[27] Raghavan Raman, Oskar van Rest, Sungpack Hong, Zhe Wu, Hassan Chafi, and Jay Banerjee. Pgx. iso: Parallel and efficient in-memory engine for subgraph isomorphism. In *Proceedings of Workshop on GRAph Data management Experiences and Systems*.

[28] Nadathur Satish, Narayanan Sundaram, Md. Mostofa Ali Patwary, Jiwon Seo, Jongsoo Park, M. Amber Hassaan, Shubho Sengupta, Zhaoming Yin, and Pradeep Dubey. Navigating the maze of graph analytics frameworks using massive graph datasets. In *ACM SIGMOD International Conference on Management of Data*, pages 979–990, 2014.

[29] Jiwon Seo, Jongsoo Park, Jaeho Shin, and Monica S. Lam. Distributed socialite: A datalog-based language for large-scale graph analysis. *Proc. VLDB Endow.*, 6(14):1906–1917, September 2013.

[30] Adam Welc, Raghavan Raman, Zhe Wu, Sungpack Hong, Hassan Chafi, and Jay Banerjee. Graph analysis: do we have to reinvent the wheel? In *First International Workshop on Graph Data Management Experiences and Systems*, page 7. ACM, 2013.

[31] Jeremiah James Willcock, Torsten Hoefler, Nicholas Gerard Edmonds, and Andrew Lumsdaine. Active pebbles: A programming model for highly parallel fine-grained data-driven computations. In *ACM SIGPLAN Notices*, volume 46, pages 305–306. ACM, 2011.

[32] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[33] Kai Zeng, Jiacheng Yang, Haixun Wang, Bin Shao, and Zhongyuan Wang. A distributed graph engine for web scale rdf data. *Proceedings of the VLDB Endowment*, 6(4):265–276, 2013.