Computation and Communication Efficient Graph Processing with Distributed Immutable View

Rong Chen†, Xin Ding†, Peng Wang†, Haibo Chen†, Binyu Zang†, Haibing Guan§
Shanghai Key Laboratory of Scalable Computing and Systems
†Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
§Department of Computer Science, Shanghai Jiao Tong University
{rongchen, dingxin, peng-wp, haibochen, byzang, hbguan}@sjtu.edu.cn

ABSTRACT

Cyclops is a new vertex-oriented graph-parallel framework for writing distributed graph analytics. Unlike existing distributed graph computation models, Cyclops retains simplicity and computation-efficiency by synchronously computing over a *distributed immutable view*, which grants a vertex with read-only access to all its neighboring vertices. The view is provided via read-only replication of vertices for edges spanning machines during a graph cut. Cyclops follows a centralized computation model by assigning a master vertex to update and propagate the value to its replicas unidirectionally in each iteration, which can significantly reduce messages and avoid contention on replicas. Being aware of the pervasively available multicore-based clusters, Cyclops is further extended with a hierarchical processing model, which aggregates messages and replicas in a single multicore machine and transparently decomposes each worker into multiple threads on-demand for different stages of computation.

We have implemented Cyclops based on an open-source Pregel clone called Hama. Our evaluation using a set of graph algorithms on an in-house multicore cluster shows that Cyclops outperforms Hama from 2.06X to 8.69X and 5.95X to 23.04X using hash-based and Metis partition algorithms accordingly, due to the elimination of contention on messages and hierarchical optimization for the multicore-based clusters. Cyclops (written in Java) also has comparable performance with PowerGraph (written in C++) despite the language difference, due to the significantly lower number of messages and avoided contention.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed programming*

Keywords

Graph-parallel Computation; Distributed Processing

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.
HPDC'14, June 23–27, Vancouver, BC, Canada.
Copyright 2014 ACM 978-1-4503-2749-7/14/06 ...$15.00.
http://dx.doi.org/10.1145/2600212.2600233 .

1. INTRODUCTION

Graph-structured computation has become increasingly popular due to its emerging adoption in a wide range of areas including social computation, web search, natural language processing and recommendation systems. With the continually increasing scale and complexity of graph dataset, it is vital to effectively express and efficiently process large-scale graph dataset, while allowing users to trivially write graph-processing programs and reason about the correctness.

The strong desire of efficient and expressive programming models for graph-structured computation has recently driven the development of several graph-parallel programming models and runtime such as Pregel [24] and its open-source alternatives [2, 1, 30], GraphLab [23, 22] and a hybrid approach called PowerGraph [12]. Basically, they encapsulate computation as vertex-oriented programs, but follow different approaches in interactions between vertices, i.e., synchronous message passing [24] vs. asynchronous shared memory [23].

As a large graph inevitably needs to be partitioned among multiple machines, we believe the following three properties are critical to a graph processing system: 1) expressiveness and programmer-friendliness so that it is not difficult to write and reason about a graph algorithm, even if it is distributed; 2) computation-efficiency so that the computation overhead is small; 3) communication-efficiency so that there won't be large amount of messages among machines and heavy contentions among messages. However, none of the above graph engines hold all the three properties. Pregel and its variants are easy to program but not computation and communication efficient as they lack support of dynamic computation and incur a lot of redundant messages. GraphLab is harder to program due to its asynchronous programming model and there is non-trivial overhead due to distributed vertex scheduling and locking. PowerGraph, which performs the best among these graph systems, requires extensive messages among machines due to distributing computation among replicated vertices.

In this paper, we describe Cyclops, a vertex-oriented graph-parallel model for distributed environment. Cyclops departs from the BSP (bulk synchronous parallel) model [32] in providing synchronous computation, but additionally introduces a key abstraction called *distributed immutable view* that provides a shared-memory abstraction to graph algorithms. To provide such a view, Cyclops replicates vertices for inter-partition edges across a partitioned graph in a cluster and only grants the master vertex with write access, whose updates will be propagated to its replicas unidirectionally at the end of each superstep (i.e., iteration). To provide computation efficiency, Cyclops grants a vertex with read-only access to all its neighbors using shared memory. As a result, a pro-

grammer can easily write a distributed graph algorithm with local semantics.

Unlike prior work (e.g., PowerGraph [12]) that distributes computation among multiple replicas of a vertex, Cyclops follows a centralized computation model such that only a master vertex does the computation and sends messages to its replicas. This is based on our evaluation that many real graphs [27, 15] do not exhibit extremely high skewed power-law distribution such that one machine cannot accommodate the computation over one vertex. Hence, it may not always be worthwhile to distribute graph computation for a single vertex among multiple machines, which causes excessive message exchanges (usually more than 5X than Cyclops, section 6.12). In contrast, there is only one unidirectional message from the replica master to each of its replicas in Cyclops, and thus there is no contention in receiving messages.

Further, being aware of the hierarchical parallelism and locality in a multicore-based cluster, Cyclops is extended with a hierarchical processing model that transparently decomposes each worker into several threads on-demand in a superstep, which is hard or impossible on the general BSP model. This significantly reduces the amount of replicas and messages within a single machine, and fully harnesses the CPU and network resources.

We have implemented Cyclops based on Hama [2], a popular open-source clone of Pregel. Cyclops mostly retains the programming interface and fault tolerance model of Hama so that most existing graph algorithms for Hama can be trivially ported to Cyclops. Our evaluation results using a set of popular graph algorithms such as *PageRank* [5], *Alternating Least Squares*, *Single Source Shortest Path*, and *Community Detection*, show that Cyclops outperforms Hama ranging from 2.06X to 8.69X on a 6-machine cluster (each machine having 12 cores and 64 GB memory) using the default hash-based graph partition algorithm. When integrating a better graph partition algorithm (i.e., Metis [20]), Cyclops achieves a significantly larger speedup over Hama, ranging from 5.95X to 23.04X. We further show that Cyclops performs comparably with PowerGraph for *PageRank* on different graphs, despite the fact that Cyclops is based on a worse baseline (execution deficiency due to managed runtime, poor object serialization and inferior RPC library). The reason is that PowerGraph has 5.5X messages compared to Cyclops.

In summary, this paper makes the following contributions:

- The *distributed immutable view* abstraction that allows efficient graph computation and distributed activation (Section 3).
- An optimization that exploits the hierarchical parallelism and locality of multicore clusters (Section 5).
- An implementation based on Hama (Section 4) and a thorough evaluation on Cyclops that confirms the efficiency and effectiveness of Cyclops (Section 6).

The rest of the paper is organized as follows. Section 2 presents an overview of BSP, and discusses issues with prior graph computation models. Section 3 describes the graph computation model of Cyclops and its overall execution flow. Section 4 describes system implementation of Cyclops, followed by the hierarchical optimization in section 5. Section 6 presents the performance evaluation results. Section 7 describes the remaining related work. Finally, we conclude the paper with a brief discussion on future work in section 8.

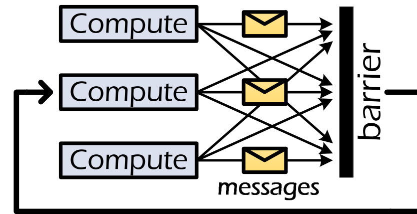

Figure 1: ***The execution flow of BSP model***

2. BACKGROUND AND MOTIVATION

This section first briefly introduces the *Bulk Synchronous Parallel* (BSP) model and the Pregel framework. Then, we examine issues with other graph programming models like GraphLab and PowerGraph.

2.1 Pregel and BSP

Pregel [24] and its open-source clones [2, 1] are built on the *Bulk Synchronous Parallel* (BSP) [32] programming model and use pure message passing to exchange updates among vertices. The Pregel framework mostly only requires programmers to provide a *compute* function for each vertex to implement a graph algorithm. Figure 1 uses a flow chart to illustrate an outline of the BSP model, which expresses the program as a sequence of *supersteps*. In each superstep, each vertex receives messages in the previous superstep from its neighbors, updates its local value using the user-defined *compute* function and sends messages to its neighbors. There is a global barrier between two consecutive supersteps where messages are aggregated and delivered to vertices. Figure 2 illustrates the pseudo-code of the *PageRank* algorithm using the BSP model. The *compute* function sums up the ranks of incoming vertices through the received messages, and sets it as the new rank of current vertex. The new rank will also be sent to its neighboring vertices by messages until a global convergence estimated by a distributed aggregator is reached or the number of supersteps exceeds a threshold.

```
public void compute(Iterator msgs) {
  double sum = 0, value;

  while (msgs.hasNext())
    sum += msgs.next();
  value = 0.15 / numVertices + 0.85 * sum;
  setValue(value);

  double error = getGlobalError();
  if (error > epsilon)
    sendMessageToNeighbors(value / numEdges);
  else
    voteToHalt();
}
```

Figure 2: ***The compute function of PageRank using the BSP model***

2.2 Issues with the BSP model

Though the BSP model has been successfully shown by Pregel to be a simple and effective alternative to handle large-scale graph processing applications, it also comes with some deficiencies in performance and accuracy. In the following, we will use the *PageRank*, the original example in the Pregel paper, on GoogleWeb dataset [27] as an example to illustrate potential issues with performance and accuracy.

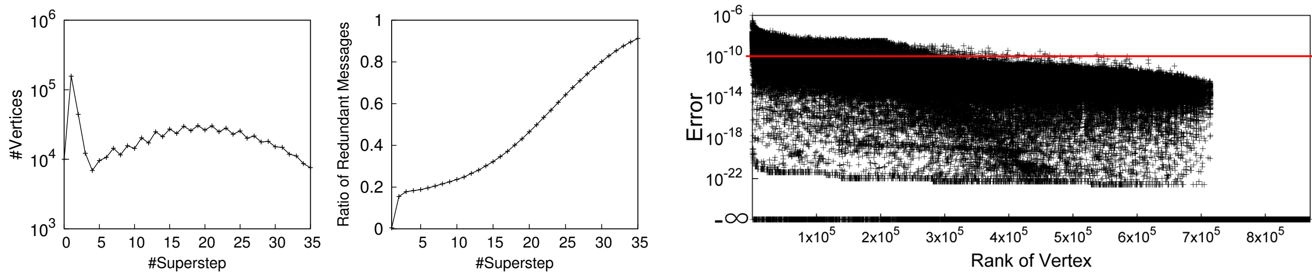

Figure 3: ***(1) Number of vertices converged in each superstep. (2) Ratio of redundant messages in each superstep. (3) Distribution of vertices error (The left ones are with higher page rank values).***

2.2.1 Computation Efficiency

The BSP model described in *PageRank* proactively pushes values by message passing in each superstep. However, some algorithms like *PageRank* are essentially pull-mode algorithm [1], where a vertex needs to collect all values from its neighboring vertices to compute the new rank. The algorithm shown in Figure 2 actually relies on the programmer to ask the framework to proactively fetch messages before the computation can continue. This unfortunately requires all neighboring vertices of a non-convergent vertex to be alive just to send (redundant) messages, even if the neighbors have converged in a very early superstep.

For many pull-mode algorithms, however, the convergence of vertices is usually asymmetric, where a majority of vertices converge in the first few iterations. Consequently, a large fraction of time is wasted to repeatedly compute over converged vertices and send messages with the same values. Figure 3(1) shows the number of vertices converged in each superstep for *PageRank* algorithm on GoogleWeb [27]. In the BSP model, about 20% vertices converge after the first two supersteps, and the majority of the vertices converge in less than 16 supersteps.

2.2.2 Communication Efficiency

The BSP model focuses on parallelizing computation using bulk synchronization, which avoids potential contention on sharing vertices. However, the communication overhead dominates the execution time in distributed environment. For example, The *PageRank* spends more than 50% execution time on sending and parsing messages in our test environment. Figure 3(2) also shows that the ratio of messages with the same value in each superstep. After 14 supersteps, there are more than 30% redundant messages in each superstep.

The communication in the BSP model allows multiple vertices to simultaneously send updates to a single vertex, which may results in contention in the receiving end. Even if the updates from the same machine can be combined, the contention from different machines is still inevitable. Further, the combiner should only be used for commutative and associative operations. Figure 4 provides an example of the communication cost for each iteration on partitioned graph for vertex 1 in different models. In the BSP model, the message enqueue operation should be protected by a *lock*. Usually, a system may use a global queue for all vertices to improve the locality of enqueue operations on batched messages, thus a *global* lock would significantly degrade the performance of communication.

[1] Informally, in a pull-mode algorithm, a vertex will proactively fetch values from its neighboring vertices to compute; a push-mode algorithm instead lets a vertex passively wait for messages from its neighboring vertices and only become active to compute upon receiving a message.

2.2.3 Convergence Detection

The termination condition of a BSP job depends on all vertices converging to an error bound and voting to halt, which means that the update of value is less than an *epsilon* defined by user. For pull-mode algorithms written in BSP, since all vertices must be alive in each superstep, the system cannot detect convergence through liveness of vertices. Hence, an application usually uses a distributed aggregator to estimate the global error (the average error of live vertices), and relies on it to determine whether to terminate the job or not.

However, there are several issues with such a convergence detection approach. First, the aggregator adds extra overhead to each superstep, since it has to apply an *aggregation* function to live vertices and gather results to the master. This may easily become a scalability bottleneck. Second, the global error is a relatively coarse-grained parameter, and thus a user cannot exactly control the proportion of converged vertices. Specifically, an algorithm using the same global error bound may get a diverse proportion of convergence with different dataset. For example, the proportion of converged vertices of the *PageRank* algorithm on Google Web and Amazon dataset [27] with the same error ($e = 10^{-10}$) is 94.9% and 87.7% respectively, according to our evaluation. Finally, as all vertices are alive in each superstep, the converged vertices may be still repeatedly computed and contribute little or even zero impact to the accumulated global error. The excessive number of converged vertices not only wastes a large number of computation and network resources, but also falsely converges some important vertices with still large error values.

Figure 3(3) shows the final errors of all vertices when the global error ($e = 10^{-10}$) is reached. The vertices are sorted by their rank values (a lower rank means a higher rank value), which means the importance of vertex. The vertices above the red line mean that they are still not converged yet. All non-converged vertices reside centrally on the upper-left corner, which is the important area due to their high rank values. A large number of vertices are with zero error values, which sink to the bottom of the figure. Hence, using global error convergence detection may cause significant accuracy issues in graph computation.

2.3 Issues with Other Models

GraphLab follows an asynchronous distributed shared memory programming model by allowing a vertex to directly read and update the values of its neighbors, which may result in relatively efficient convergence [4]. However, programmers need to understand the consistency model, and the execution on GraphLab is non-deterministic in essence, making it hard to debug and diagnose correctness and performance bugs. Further, the performance overhead due to distributed vertex scheduling and locking for consistency may reduce the improvement from the elimination of the global barrier and allowing direct vertex access using shared mem-

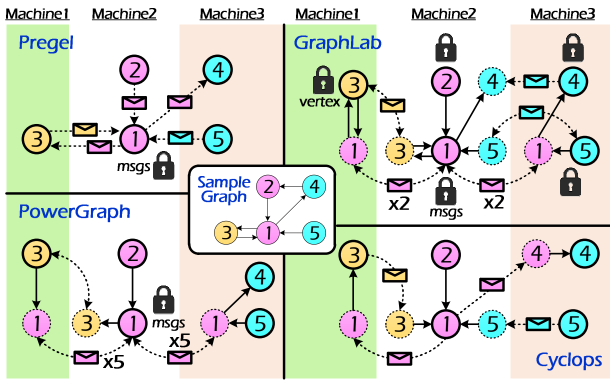

Figure 4: ***An example of communication cost in different models***

```
public void compute() {
  double sum = 0, value, last = getValue();

  Iterator *edges = getInEdgesIterator();
  while (edges.hasNext())
    sum += edges.next().vertex.getMessage();
  value = 0.15 / numVertices + 0.85 * sum;
  setValue(value);

  double error = Math.abs(value - last);
  if (error > epsilon)
    activateNeighbors(value / numEdges);
  voteToHalt();
}
```

Figure 5: ***The compute function of PageRank in Cyclops***

ory. As shown in Figure 4, to compute over vertex 1, it first *locks* itself and then asks all its neighboring vertices (may through replicas) to be *locked* using distributed vertex locks before computation. Finally, GraphLab enforces all operations on vertices in a pure local fashion, thus it requires to create duplicate replicas for each edge spanning machines and demands bidirectional messages (i.e., sending update from master to replicas and activation from replicas to master). In Figure 4, the edge from vertex 1 to vertex 4 appears in both machines and incurs two replicas. The replica 4 implements local activation for vertex 1, while the replica 1 implements local access for vertex 4. Due to bidirectional communication, vertex 1 may receive multiple activation messages from its replicas. Hence, there may be contention on vertex 1 and it requires a *lock* to coordinate message receiving.

PowerGraph [12] abstracts computation as *Gather*, *Apply* and *Scatter* (GAS), in which a vertex collects messages from its replicas, computes its new value and sends the new value to all its replicas, and ask all its replicas to activate their neighboring vertices. As shown in Figure 4, it takes three rounds of bidirectional message passing between a master and its replicas in each iteration to support the GAS model. The major benefit is that this can decompose the computation of an extremely skewed vertex in natural graphs to multiple machines. However, the bidirectional message passing between a master and replicas also results in contention when multiple replicas send messages to the master in the Gather and Scatter phase. Further, the GAS model requires about 5 messages for each replica of the vertex in one iteration (2 for Gather, 1 for Apply and 2 for Scatter), which significantly degrades the performance.

3. DISTRIBUTED IMMUTABLE VIEW

Being aware of the deficiency with prior systems, we describe Cyclops, a synchronous graph processing model that departs from the BSP model implemented in Pregel, and combines the best features from both GraphLab and PowerGraph. From GraphLab, Cyclops borrows direct memory access through vertex replicas to avoid redundant computation and messages. From PowerGraph, Cyclops borrows distributed activation to avoid duplicate replicas and bidirectional communication.

At the heart of Cyclops is the *distributed immutable view* abstraction, which presents a graph application with the view of the entire graph right before the beginning of each superstep. Unlike Pregel that requires message passing to push updates to neighboring vertices, the view grants a vertex with read-only access to its neighboring vertices through shared memory, thus providing local semantics to programmers. The immutable view abstraction still retains the synchronous and deterministic nature of the BSP model. Unlike GraphLab that limits replicas to single purpose (i.e., access or activation), the replicas in distributed immutable view also bear the task of distributed activation of vertices, thus avoiding the messages from replicas to its master. Hereby, *distributed immutable view* only requires one round one-way message from master to its replicas, and thus is immune from contention among messages.

In the rest of this section, we will use the *PageRank* algorithm as a running example and describe the programming interface, graph organization, vertex computation, message passing and execution model in Cyclops.

3.1 Programming Interface

Cyclops mostly retains the programming interface of BSP implemented in Pregel (see Figure 2). The key difference is that instead of using message passing to receive updates from its neighboring vertices, Cyclops relies on shared memory access to directly read the values from its neighboring vertices. Further, Cyclops uses a local error detection scheme instead of using the average error from all vertices, thus a vertex will deactivate itself by default and only become active again upon receiving activation signal.

Figure 5 shows an example implementation of the *compute* function of *PageRank* in Cyclops. In contrast to the implementation in Figure 2, the iterator of messages is no longer necessary, but instead the application directly reads values from neighboring vertices provided by the *distributed immutable view*. Hence, Cyclops no longer requires keeping all vertices alive for sending messages. Further, it no longer relies on the global error but instead uses the local error to decide whether to activate neighboring vertices. By default, a vertex will deactivate itself and only become active again upon receiving activation signal at the end of each superstep.

3.2 Graph Organization

The program state in the BSP model is modeled as a directed graph, which is split into multiple partitions and assigned to multiple workers. Each worker is in charge of running the *compute* function on local vertices in parallel. Since a graph is partitioned, the communication between vertices is performed by message passing.

Similarly, the graph in Cyclops is also split and assigned to workers. However, to provide a *distributed immutable view* for each vertex, Cyclops maintains a read-only replica of each vertex for edges spanning machines during a graph cut. This makes sure that there is always a read-only memory reference for a partitioned graph in each machine. In each superstep, only the master vertex may be scheduled to execute the *compute* function, while replicas are just one-way synchronized by its master vertex at the end of each superstep.

Figure 6 shows a sample graph with six vertices, which is split and assigned to three workers. In the BSP model (Part A), for example, the worker 2 would run the *compute* function on vertex 3

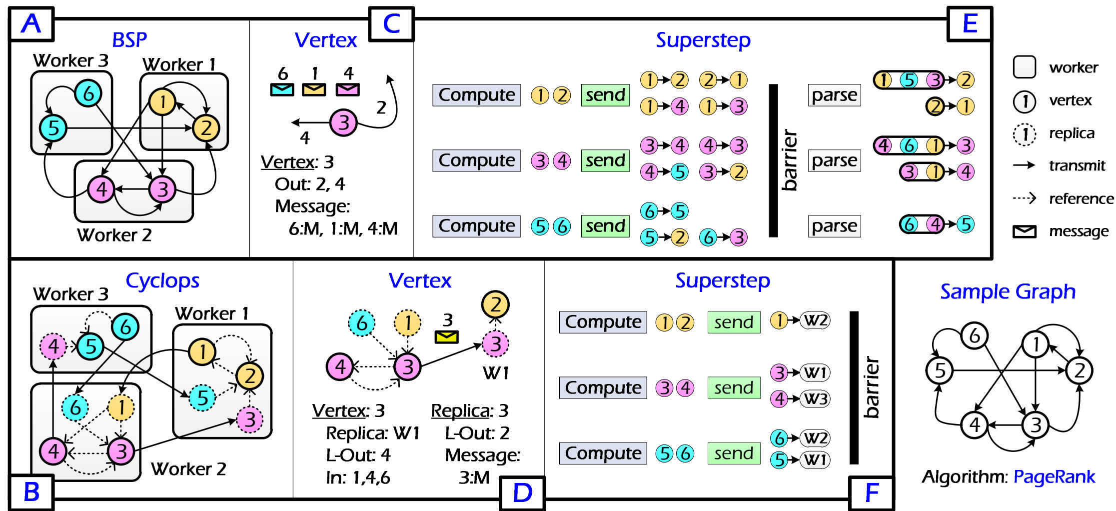

Figure 6: ***An example of PageRank algorithm on Pregel and Cyclops for a simple graph. The symbol Out and In mean out-edges and in-edges, and the L-Out represents out-edges to local vertices. The message from vertex X is label as X:M.***

with messages from vertices 1, 4 and 6, and send two messages to vertices 2 and 4. In contrast, the *compute* function on vertex 3 in Cyclops (Part B) may directly read data from vertex 4 and replicas 1 and 6, and synchronize data with replica 3 in worker 1.

3.3 Vertex Computation

Vertex computation in the BSP model is presented in the form of user-defined *compute* functions. The *compute* function can be executed in parallel, as it is only allowed to inspect and modify the data of the current vertex, including the value and edges. In addition, all external information is obtained by messages as parameters, which should be sent in the previous superstep. Accordingly, the messages sent by the *compute* function will arrive before the forthcoming superstep.

Rather than using message passing, Cyclops provides a *distributed immutable view* of the previous superstep to each vertex like the *scope* in GraphLab [22]. However, the immutable view is synchronous, and thus the *compute* function can freely access it in read-only mode without worrying about consistency issues. Based on the immutable view, Cyclops naturally implements dynamic computation to support pull-mode algorithms. The *compute* function can directly access values of neighbors, even if they have converged and are inactive.

As shown in Figure 6, vertex 3 in the BSP model (Part C) maintains information (*e.g.*, the unique identifier) of outgoing neighbors (vertices 2 and 4) for sending messages, and messages from incoming neighbors (vertices 1, 4 and 6). In Cyclops (Part D), the references of incoming neighbors (vertices 1, 4 and 6) are stored to in-edges of vertex 3, which provide an immutable view to the *compute* function on vertex 3. Note that the references of vertices 1 and 6 are pointed to the replicas, since they are the remote vertices. The rest edges of vertex 3 are used to activate local vertices and synchronize with its replicas.

3.4 Message Passing

In the BSP model, message passing is used both to transfer data and to activate vertices. It results in the contention on message enqueue, and a large number of redundant computation and message passing for converged vertices in pull-mode algorithms.

In Cyclops, *the data movement between adjacent vertices is decoupled from message passing* as data transfer between them is performed by shared memory access. Cyclops uses a *distributed* approach to implement vertex activation by using a master vertex to send activation requests together with values to propagate to its replicas. As the remote worker with outgoing neighbors of current vertex must have a replica of vertex, each vertex and its replicas are responsible for activating its local outgoing neighbors.

The only message required in Cyclops is used to synchronize replicas with their master vertices in each superstep. It guarantees each replica only receiving at most one message, thus there is no protection mechanisms in message passing of Cyclops. For the sample graph in Figure 4, all messages could be served in parallel in Cyclops.

In Figure 6, the out-edges of vertex 3 in the BSP model (Part C) are used to send message to outgoing neighbors (vertices 2 and 4) regardless of whether they are in local or remote workers. In Cyclops (Part D), vertex 3 maintains the location (worker 1) of replica for synchronization. The local out-edges to vertices 4 and 2 maintained in vertex 3 and its replica are used for distributed activation.

3.5 Execution Model

The BSP execution model uses a separate ingress phase to load and split a directed graph from the underlying file system. The input graph is split by a distributed graph partitioning heuristic (*e.g.*, random hashing, Metis [20]) into multiple partitions and assigned to workers. The execution phase of the BSP model presents as a single loop of *supersteps*. Each superstep consists of four sequential operations: message parsing (PRS), vertex computation (CMP), message sending (SND) and global barrier (SYN). At the beginning of each superstep, while there are messages or vertices alive, the worker parses messages received in the last superstep and uses them to activate vertices. After that, all active vertices execute user-defined *compute* function, and send messages to neighbors. A vertex deactivates itself by voting to halt. Before entering the global barrier, all messages sent in current superstep should be transmitted to destination.

Cyclops follows a similar execution model. Each worker executes the *compute* function on active master vertices, which pulls data from neighbors through shared memory access. The modification on non-converged vertices results in synchronization messages from master to replicas. Because the messages are directly used to update replicas and activate local neighbors in parallel by receiv-

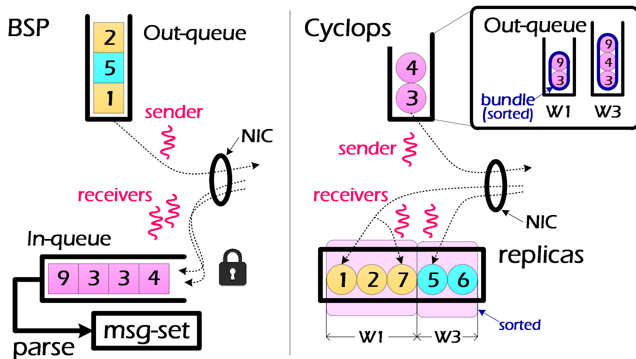

Figure 7: *An example of communication in BSP and Cyclops*

ing threads, Cyclops does not require the message parsing before vertex computation.

Figure 6 illustrates four consecutive operations intercepted from two connected supersteps. The two execution flows are similar (Part E and F), except for messages. The workers in the BSP model send and parse messages from vertices to its outgoing neighbors (*e.g.*, from vertex 3 to vertices 2 and 4). In contrary, the worker in Cyclops only sends the messages and update the replicas (*e.g.*, vertex 3 to replica 3).

3.6 Fault Tolerance

The fault tolerance mechanism used in Pregel is based on checkpoint and restore. After each global barrier, workers can save the states of their graph partition to underlying storage layer (e.g., HDFS). The necessary state consists of superstep count, vertex values, edge values and messages. This is much simpler than an asynchronous shared memory system like GraphLab, which requires an eventual consistent checkpoint algorithm [8]. Cyclops follows a similar mechanism used in Pregel, except that workers does not require to save the replicas and messages.

4. IMPLEMENTATION ISSUES

We have implemented Cyclops and its optimizations based on Apache Hama [2], a popular clone of Pregel implemented in Java. Cyclops adds around 2,800 SLOCs to Hama. Cyclops is mostly compatible with the interfaces and graph processing in Hama. However, there are a few differences to support the *distributed immutable view* in Cyclops.

4.1 Message Passing

Hama splits the vertex computation and message passing to avoid interference. As shown in Figure 4, all messages are cached in a global out-queue before sending. To improve the network utilization, Hama combines the messages sent to the same vertex if possible, and bundles the messages sent to the same worker in one package. Further, Hama uses a global in-queue to temporally store all messages to exploit locality of enqueue operations, and parses messages to each vertex at the beginning of next superstep. The enqueue operations from multiple receiving threads should be *serialized* to avoid contention.

In Cyclops model, the replica only receives at most one message, thus we optimize message passing to directly update replicas in parallel by multiple receiving threads. The message combining and parsing are no longer necessary. To further improve the locality, Cyclops groups replicas according to the location of its master, and sorts replicas within group at graph ingress. Cyclops uses multiple sub-queues to separately cache messages sent to different workers, and sorts the messages in bundle before sending.

4.2 Graph Partition

Cyclops is orthogonal to the graph partition algorithms and the default partition algorithm (i.e., hash partition) can be used directly without changes to Cyclops. However, as the graph partition quality may affect the amount of replicas in Cyclops, using a better partition algorithm may generate a balanced edge-cut. This may evenly assign vertices to workers and reduce the number of inter-partition edges, thus reduces the amount of replicas and replica synchronization messages. Hence, we additionally implement the Metis [20] partition algorithm that tries to minimize inter-partition edges and balance the vertices among partitions. In section 6.6, we will show that this may result in significant performance boost.

4.3 Graph Ingress

The in-memory graph organization is slightly different from that in Hama due to the need for creating replicas, adding in-edges and local out-edges for all vertices to maintain the *distributed immutable view* for the *compute* function. To avoid unnecessary burden for programmer, Cyclops maintains compatibility of the input file format with Hama, but instead reuses the ingress phase to load and split graph.

In addition to the ingress phase in Hama, Cyclops adds its own ingress phase to create replicas and add in-edges and local out-edges. This is done by letting each vertex to send a message to its out-edges. Each vertex will create a replica for the sending vertex upon receiving a remote message when such a replica is not created yet. It will further create an in-edge from the replica and a local out-edge for the replica to itself. This is essentially a superstep in Hama.

4.4 Convergence Detection

Cyclops supports the original global aggregation based convergence detection. However, as we discussed in section 2.2.3, using such a global error to indicate whether the graph processing have converged may cause accuracy problems. Hence, we further add a fine-grained convergence detection scheme by counting the proportion of converged vertices, which is more suitable for the dynamic computation nature of many graph algorithms.

5. HIERARCHICAL GRAPH PROCESSING WITH CYCLOPS

With the prevalence of multicore-based clusters, the two-level hierarchical organization raises new challenges and opportunities to design and implement an efficient large-scale graph processing framework. To our knowledge, Pregel and its open-source clones like Hama currently are oblivious to the underlying hardware topology and use a uniform way to manage all workers.

Inspired by the hierarchical-BSP model [6], we apply a hierarchical design to exploit such parallelism and locality, which is called CyclopsMT. CyclopsMT uses a three-level scheme to organize tasks: the main task in superstep (level 0) is first partitioned into machines in cluster (level 1), and the worker in each machine further partitions the task to multiple threads running on multicore hardware (level 2). At any time, only the last-level threads perform tasks. Their parents, i.e., the workers in higher levels, just wait until all their child threads finish their tasks. Note that, CyclopsMT still preserves the synchronous and deterministic computation nature in Cyclops, as all threads sharing the same parent synchronize with each other at the end of each superstep.

However, it is non-trivial to parallelize tasks in each superstep using multi-thread for BSP model. This is because the message operations in each superstep have *poor locality* and *heavy contention*.

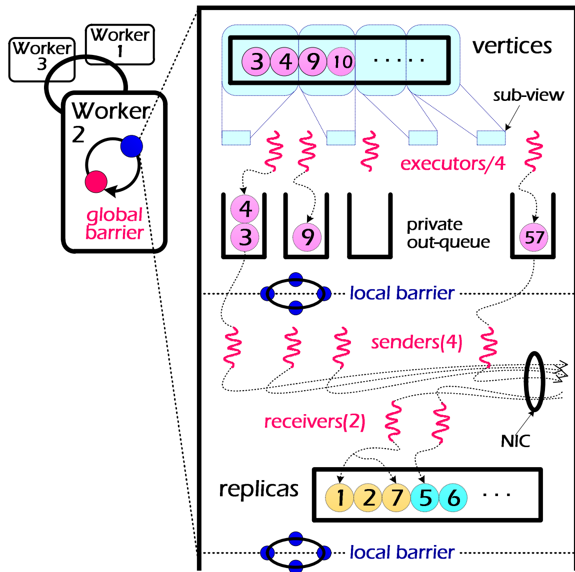

Figure 8: ***The architecture of hierarchical Cyclops (CyclopsMT)***

For message parsing, trivially splitting received messages to multiple threads may result in heavy contention on destination vertices, especially when a large number of messages are delivered to the same vertex. For message sending, using a separate queue to buffer messages for each thread is harmful to message batching and aggregation.

In Cyclops, fortunately, the data movement between vertices is decoupled from message passing, and thus the messages are only sent from master vertices to their replicas and there are no duplicate replicas in the destination. This opens opportunities of parallelizing each superstep in Cyclops. Hence, CyclopsMT adopts a *split* design to parallelize computation on master vertex and message passing to replicas in each superstep, and to exploit the locality of communication and synchronization without contention.

Figure 5 illustrates how CyclopsMT parallelizes all operations in a superstep. For vertex computation, all vertices are evenly mapped to multiple threads and are executed fully in parallel. The vertex activation requests in the *compute* function are treated differently. The remote activation is delayed to message sending during replica synchronization, while the local activation is performed immediately by setting the corresponding incoming edge of destination vertex, which is a *lock-free* operation.

In the message sending phase, all updated vertices with replicas need to send synchronization messages. The remote activation requests will be combined with synchronization messages. Since there are no duplicate messages sent to the same destination replica, message combination is no longer required. *Private* out-queues are used to reduce the contention on underlying network hardware.

In the message receiving phase, multiple message receivers are launched to exploit the parallelism. Because there are no duplicate messages sent to the same replica, the update operation on replica is *lock-free* and *non-blocking*. However, with the growing number of threads within one worker, too many message receivers would result in heavy contention on underlying network hardware. The improvement is also devoured by the workload imbalance of message receivers.

CyclopsMT support separately configure the parallelism of vertex computation and message passing according to different behavior of algorithm and workloads. In Figure 5, CyclopsMT launches

Table 1: ***A collection of real-world graphs.***

Algorithm	Graph	$\|V\|$	$\|E\|$
PageRank	Amazon	403,394	3,387,388
	GoogleWeb(GWeb)	875,713	5,105,039
	LiveJournal(LJournal)	4,847,571	69,993,773
	Wiki	5,716,808	130,160,392
ALS	SYN-GL	110,000	2,729,572
CD	DBLP	317,080	1,049,866
SSSP	RoadCA	1,965,206	5,533,214

4 working threads to compute master vertices in parallel, and 2 message receivers to receive messages and update replicas.

Finally, with the growing number of participants, the performance overhead of global barrier rapidly increases. Hierarchical design in CyclopsMT provides a natural solution to reduce the overhead. The main thread represented as the whole worker performs distributed protocol of a global barrier, and the rest of threads wait on a local barrier. The hierarchical barrier reduces the number of messages and the latency of communication.

6. EVALUATION

This section evaluates Cyclops and its optimizations against the baseline system Hama using four typical graph algorithms: *PageRank* (PR), *Alternating Least Squares* (ALS), *Community Detection* (CD) and *Single Source Shortest Path* (SSSP). The first three are pull-mode algorithms, while the fourth one is a push-mode algorithm.

6.1 Overview of Tested Graph Algorithms

PageRank [5] (PR): It is a widely-used and well-studied graph algorithm. A web page's rank is computed as a weighted sum of all its incoming neighbors' rank value. In graph-parallel models, each vertex receives messages from all its incoming neighbors to compute its new rank and send the new value to all its outgoing neighbors. Since a vertex needs to gather data from all its neighbors, PageRank is a typical pull-mode algorithm.

Alternating Least Squares (ALS): It was used by Zhou et.al [36] to do recommendation in Netflix. The input to ALS is a sparse users by movies matrix R, where each entry contains the movie rating of each user. This algorithm uses U * R to simulate the ranking value. It iteratively refines U and V by computing the least square solution with the other fixed. ALS can easily fit into the graph computation framework if we consider the input matrix as a graph connecting users with movies [12].

Community Detection (CD): It is a simple community detection application based on label propagation [36]. Each vertex has a label value, which is assigned with the most frequent labels in its neighbors. Vertices with the same label are considered as a community.

Single Source Shortest Path (SSSP): It is a typical push-mode application. A vertex will not do computation unless messages arrive to wake it up. A vertex uses only the incoming messages to update its value. After that, it can go to sleep. We use the SSSP algorithm to show that even the push-mode applications has no redundant vertex computation and message passing, Cyclops and CyclopsMT still outperforms Hama through the elimination of contention on communication and exploiting the parallelism and locality of multicore-based cluster.

All these algorithms can be trivially written/ported to Cyclops due to its synchronous nature with local semantics exported to the *compute* function. It requires 8 and 7 SLOCs to adapt the existing Hama implementation of PR and SSSP to Cyclops. We implement ALS and CD to both Hama and Cyclops and the code difference between Hama and Cyclops is only 10 and 6 SLOCs accordingly.

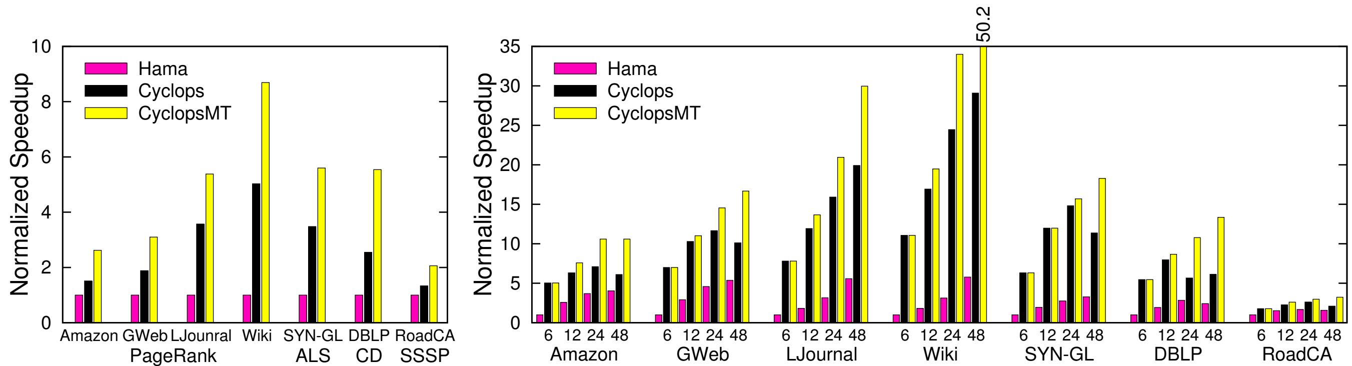

Figure 9: *(1)The speedup of Cyclops and CyclopsMT over Hama using 48 workers for a variety of dataset. (2)The scalability of Cyclops and CyclopsMT over Hama using 6, 12, 24 and 48 workers for a variety of dataset (Speedup is normalized against Hama with 6 workers).*

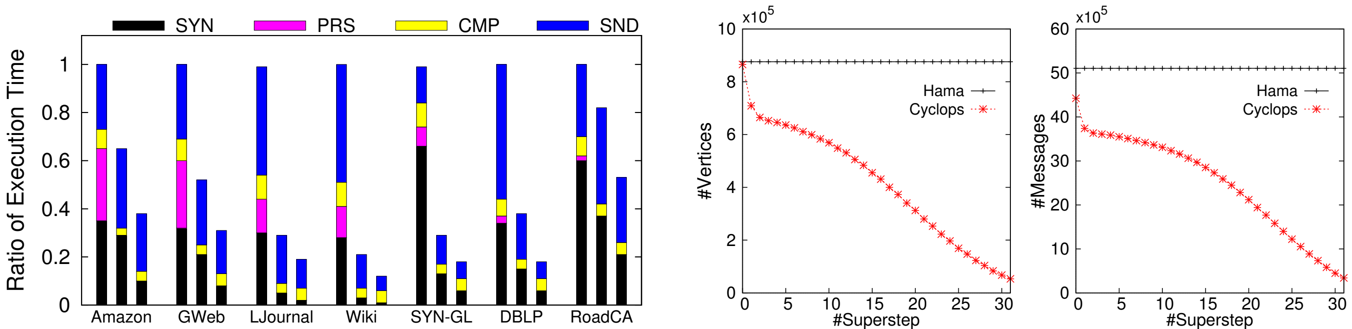

Figure 10: *(1) The breakdown of execution time on Hama, Cyclops and CyclopsMT using 48 workers with different benchmarks. (2) The number of active vertices in each superstep on Hama and Cyclops for PageRank algorithm with GWeb dataset. (3) The number of messages in each superstep on Hama and Cyclops for PageRank algorithm with GWeb dataset.*

6.2 Experimental Setup

All experiments are performed on an in-house 6-machine multicore cluster. Each machine has a 12-core AMD Opteron 6168 CPU, 64GB RAM, 2x1TB Hard Drivers and 1 GigE network ports. We run HDFS on the same cluster for underlying storage layer, and use a graph ingress phase to load the graph to main memory before doing graph computation.

Table 1 lists a collection of large graphs used in our experiments. Most of them are from Stanford Large Network Dataset Collection [27]. The Wiki dataset is from [15]. The dataset for the ALS algorithm is synthetically generated by tools provided from that used in the Gonzalez et al. [12]. The SSSP algorithm requires the input graph to be directed and weighted. Since the RoadCA graph is not originally weighted, we synthetically assign a weight value to each edge, where the weight is generated based on a log-normal distribution ($\mu = 0.4, \sigma = 1.2$) from the Facebook user interaction graph [34].

6.3 Overall Performance

Figure 9 summarizes the normalized performance of Cyclops and CyclopsMT compared to Hama with different configurations. Note that we evenly distribute workers in each machine, and run 8 threads at most on a single machine for CyclopsMT because JVM doesn't scale well on large-scale multicore platforms and the JVM itself will create a number of auxiliary threads. The number of workers shown in figure for CyclopsMT is equal to the total number of threads.

As shown in Figure 9(1), Cyclops and CyclopsMT outperforms Hama in all algorithms over different datasets with 48 workers. For PR, the speedup increases with the growing size of input graph. The largest speedup of Cyclops and CyclopsMT comes from Wiki (our biggest graph), which is 5.03X and 8.69X accordingly. The speedup of Cyclops and CyclopsMT is also remarkable for ALS with SYN-GL (3.48X and 5.60X) and CD with DBLP (2.55X and 5.54X). For SSSP with RoadCA, the performance improvement is relative small (1.33X and 2.06X). This is because SSSP is a typical push-mode algorithm, and thus there are no redundant vertex computation and message passing.

Figure 9(2) compares the scalability of Cyclops and CyclopsMT compared to Hama with different workers from 6 to 48. With the growing of workers, the number of edges spanning machines rapidly increases, which results in the amplification of the number of messages. Hence, the time spent on message passing in several applications significantly degrades the performance of Hama and Cyclops. However, due to exploiting the locality of messages and using hierarchical barrier, CyclopsMT reduces the performance degradation in communication. Further, for applications whose performance are dominated by vertex computation, the scalability is still quite good, including PR with most graphs, ALS and CD. For the performance of application dominated by communication, the growth of speedup appears slightly slowdown, including PR with Amazon and SSSP with RoadCA.

6.4 Performance Breakdown

In this section, we categorize the source of performance speedup through a breakdown of execution time and the number of active vertices and messages in each superstep.

Figure 10(1) shows the ratio of execution time breakdown of benchmarks on Hama, Cyclops and CyclopsMT using 48 workers with different benchmarks. The result is normalized to Hama, and the labels SYN, PRS, CMP and SND correspond to synchro-

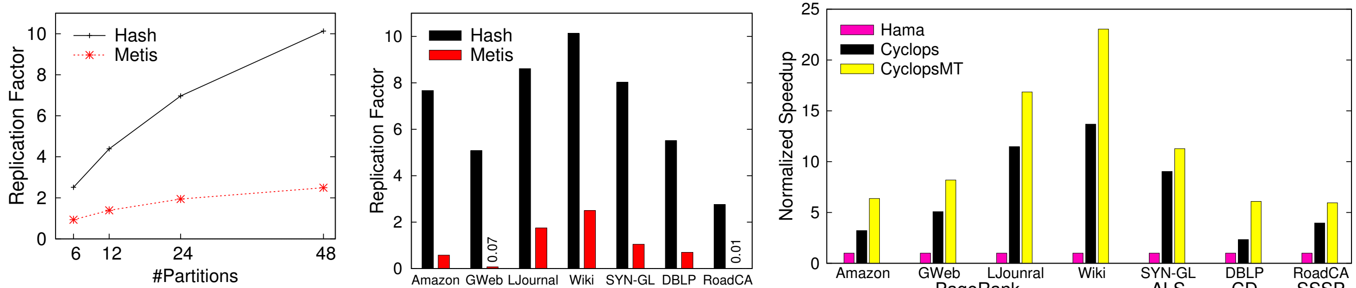

Figure 11: *(1) The replication factor on Wiki dataset using different partitions. (2) The replication factor for a variety of dataset using 48 partitions. (3) The performance of Hama, Cyclops and CyclopsMT with Metis partition using 48 workers for a variety of dataset. Speedup is normalized against Hama under Metis partition.*

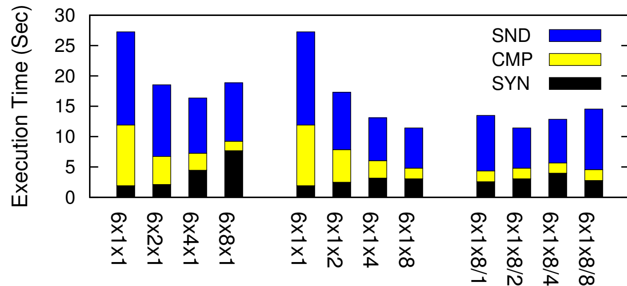

Figure 12: *The breakdown of execution time on CyclopsMT for PageRank with GWeb dataset using different configurations. The labels under histogram are the configuration, and the symbol of 'MxWxT/R' corresponds to #machines, #workers, #threads and #receivers.*

nization, message parsing, vertex computation and message sending accordingly. Cyclops and CyclopsMT significantly outperform Hama for pull-mode applications because of two major benefits. The first is from the elimination of redundant computation and message passing for converged vertices, which efficiently improves the performance of pull-mode applications. The second exploits the parallelism and locality of message passing, through the elimination of the contention in message parsing. The significant improvement in SYN phase also benefits from the load balance in each superstep. CyclopsMT further reduces the number of replicates and messages within a single machine, and improves the synchronization among workers by the hierarchical barrier. For SSSP, Cyclops and CyclopsMT achieves modest speedup to Hama through optimized message passing and efficient vertex access through shared memory.

Figure 10(2) and (3) show the number of active vertices and messages in each superstep for PageRank algorithm on GWeb. The number of active messages and messages decides vertex computation time and message passing time respectively. Compared to Hama, Cyclops significantly reduces the number of active vertices and messages as expected.

6.5 Improvement of CyclopsMT

To illustrate the effect of hierarchical Cyclops (CyclopsMT) on multicore-based clusters, we evaluate the PR algorithm on GWeb dataset with different configurations of CyclopsMT. In Figure 12, the configuration labeled 6xWx1 correspond to Cyclops, which launch W single thread workers on each machine. With the increase of workers, the workload of vertex computation is constant, but the number of messages increases because of the growing number of replicas. Cyclops can efficiently parallelize the vertex computation and message parsing, thus the execution time of computation (CMP) rapidly decreases and the communication time(SND) is stable. Further, the overhead of synchronization (SYN) increases with the growing number of participants, which results performance degradation when the number of workers exceeds 24. The configuration labeled 6x1xT correspond to CyclopsMT, which launch 1 worker with T threads on each machine. Because of the fixed number of worker, the overhead of communication and synchronization are also stable. The performance improvement mainly comes from vertex computation. The only contention in CyclopsMT is from the underlying hardware. Too many receiving threads would contend on CPU and network resources, thus CyclopsMT provides separate configuration to control interference. The best performance is from configuration labeled 6x1x8/2, which only launches 2 receiving threads for communication.

6.6 Impact of Graph Partitioning Algorithms

In all prior evaluation, we simply use the naive hash-based graph partitioning algorithm, which may result in an excessive amount of edges being cut. To study the impact of graph partition algorithm on Hama and Cyclops, we compare the performance using two graph partitioning methods (*e.g.*, Hash-based and Metis [20]) with all algorithms on different datasets. Figure 11(1) depicts the average number of replicas for different number of partitions for Wiki dataset and Figure 11(2) shows the average number of replicas for different datasets on 48 partitions, using the hash-based and Metis partition algorithms. With the increase of partitions, the average number of replicas under the hash-based partition algorithm rapidly approaches the average number of edges per vertex. In contrast, Metis significantly reduces the average number of replicas.

In Figure 11(3), Cyclops and CyclopsMT using Metis significantly outperform their counterparts using hash-based partitions. However, Hama does not obviously benefit from the Metis partition algorithm, because the Metis partition algorithm only tries to minimize the total number of edges spanning machines while trying to balance vertices, and the vertices may be a little bit out of balance. In Cyclops, as the number of converged vertices rapidly increases along with graph computation, the degree and impact of imbalance for vertices will be decreased. However, Hama will remain imbalanced along all of its execution.

6.7 Ingress Time

The input graph is loaded from a text file stored in a distributed file-system (HDFS). The graph processing runtime then splits the file into multiple blocks and generates in-memory data structures by all workers in parallel. Each worker reads vertex from a block and sends vertices to their target workers according to the graph

Graph	LD (sec)	REP (sec)	INIT (sec)	TOT (sec)
	H / C	H / C	H / C	H / C
Amazon	6.2 / 5.9	0.0 / 2.5	1.7 / 1.5	7.9 / 9.9
Gweb	7.1 / 6.8	0.0 / 2.8	2.6 / 1.9	9.7 / 11.4
Ljounral	27.1 / 31.0	0.0 / 44.7	17.9 / 9.2	45.0 / 84.9
Wiki	46.7 / 46.7	0.0 / 62.2	33.4 / 20.4	80.0 / 129.3
SYN-GL	4.2 / 4.0	0.0 / 2.6	2.4 / 1.8	6.6 / 8.4
DBLP	4.1 / 4.1	0.0 / 1.5	1.3 / 0.9	5.4 / 6.5
RoadCA	6.4 / 6.2	0.0 / 3.9	0.9 / 0.6	7.3 / 10.7

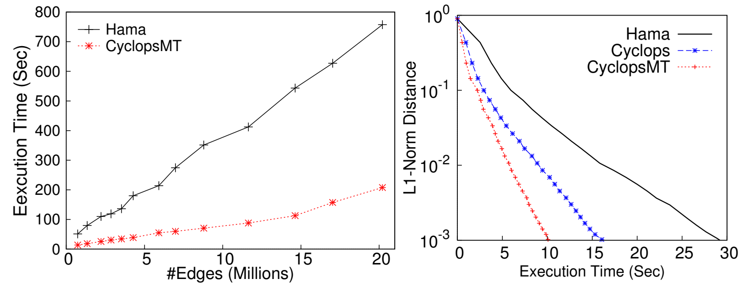

Figure 13: ***(1)The execution time of graph ingress. H represents Hama, and C represents Cyclops (2)The scalability of Cyclops with ALS benchmark. (3)The L1-Norm distance of Hama, Cyclops and CyclopsMT on PageRank algorithm with GWeb dataset.***

Table 2: ***The memory behavior of Hama, Cyclops and CyclopsMT on PageRank algorithm with Wiki dataset.***

Configuration	Max Cap (GB)	Max Usage (GB)	Young GC		Full GC	
			Num	Sec	Num	Sec
Hama/48	1.7	1.5	132	45.7	69	18.7
Cyclops/48	4.0	3.0	45	62.9	15	13.9
CyclopsMT/6x8	12.6/8	11.0/8	268/8	67.8/8	32/8	2.52/8

partition algorithm. Finally, each worker initializes its own vertices. Cyclops requires an additional phase to create replicas and refine vertices.

Figure 13(1) shows the breakdown of ingress time using 48 workers with different input graphs. We split the ingress time into graph loading (LD), vertex replication(REP) and vertex initialization (INIT). The overhead of ingress time in Cyclops is mainly from the vertex replication phase, and the rest portion of time is close to Hama. The time spent on vertex replication depends on the size of graph, and the increase of time is still modest. Nevertheless, this is a one-time cost as a loaded graph will usually be processed multiple times.

6.8 Scale with Graph Size

To show how CyclopsMT scales with graph size, Figure 13(2) illustrates the execution time of ALS algorithm on dataset with a varying number of edges from 0.34M to 20.2M using 48 workers. The execution time only increases from 9.6 (for 0.34M) to 207.7 (for 20.2M) seconds, indicating that the performance of CyclopsMT scales well with the growing size of input graph.

6.9 Convergence Speed

To evaluate the convergence speed of Cyclops and CyclopsMT compared to Hama, we evaluate the L1-Norm distance to the final result as the execution time goes by. The final result is collected offline, and the partial result of applications on Cyclops, CyclopsMT and Hama are dumped after each superstep. In Figure 13(3), both Cyclops and CyclopsMT significantly accelerate the convergence of PageRank on GWeb dataset compared to Hama.

6.10 Memory Consumption

As Cyclops needs some replicas for vertices to maintain the *distributed immutable view*, an intuitive impression is that Cyclops may significantly increase the memory consumption. To compare the memory behavior of Cyclops and Hama, we use *jStat* to evaluate the memory usage and the times of the garbage collection (GC) execution. Note that we configure the Hama and Cyclops are configured using Concurrent Mark-Sweep (CMS) collector as GC. The partition algorithm is the hash-based partition instead of Metis, which should be a worst case of memory consumption for Cyclops.

Table 2 illustrates the memory behavior per worker of Hama, Cyclops and CyclopsMT using 48 workers for the PageRank algorithm. The input graph is Wiki, and CyclopsMT is configured as 1 worker per machine with 8 threads. The maximum memory spaces allocated to Cyclops is larger than that in Hama. However, The number of Young and Full GC in Cyclops is actually less than Hama due to the elimination of redundant messages, which occupies a large number of memory in each superstep. CyclopsMT overall consumes much less memory per work than Cyclops and Hama, since it shares replicas among threads, and replaces the usage of internal message with memory reference.

Table 3: ***Message passing micro-benchmark results.***

#Message	Hama (sec)			PowerGraph (sec)			Cyclops (sec)
	SND	PRS	TOT	SND	PRS	TOT	
5M	9.7	0.4	10.1	0.7	0.1	0.8	1.0
25M	56.4	1.9	58.3	3.4	0.2	3.6	5.6
50M	183.4	3.8	187.2	6.9	0.4	7.3	9.6

6.11 Communication Efficiency

We demonstrate the benefits of unidirectional communication of Cyclops by comparing the results of the message passing micro-benchmark using three different implementations. The micro-benchmark launches five workers to concurrently send messages to update the element of an array in master worker. Each message is a pair of index and value. The first implementation used by Hama is based on Apache Hadoop RPC lib. It uses a global queue to serially buffer messages from multiple senders, and uses an additional message parsing phase to update values to array. The second implementation used by PowerGraph is based on C++ Boost RPC lib, and it adopts the same method to send and parse messages. The last implementation used by Cyclops is also based on Apache Hadoop RPC lib, but it directly updates the messages from multiple senders to array without protection. As shown in Table 3, there is one order of magnitude performance slowdown between the implementations on Hama and PowerGraph, even using the same method. However, the implementation used by Cyclops has slightly better performance than PowerGraph, even if Cyclops uses a much worse RPC library as that in Hama, due to significantly less messages (see next section).

6.12 Comparing with PowerGraph

Since PowerGraph [12] is the well-known distributed graph processing system, readers might be interested in how the performance of Cyclops compares to that of PowerGraph, even if PowerGraph

Table 4: ***A comparison between CyclopsMT and PowerGraph.***

DataSet	Hash-based Partition						Heuristic Paritition		
	Execution Time(s)		AVG #Replicas	#Messages(M)	Msg/Rep	CMP	Execution Time(s)		AVG #Replicas
	Cyclops : PG	Net	Cyclops : PG	Cyclops : PG	Cyclops : PG	Cyclops	Cyclops : PG	Net	Cyclops : PG
Amazon	10.5 : 14.8	+41%	3.86 : 3.77	38 : 192	1.0 : 5.2	11%	4.9 : 7.8	+59%	0.24 : 0.40
GWeb	11.4 : 15.2	+33%	2.44 : 2.57	38 : 212	1.0 : 5.3	15%	4.9 : 6.5	+33%	0.04 : 0.82
Ljournal	97.1 : 72.9	-25%	2.69 : 2.62	353 : 1873	1.0 : 5.4	25%	53.0 : 49.1	-8%	0.64 : 1.18
Wiki	75.6 : 61.9	-18%	2.51 : 2.60	218 : 1366	1.0 : 6.2	39%	59.9 : 43.2	-28%	0.93 : 1.08

was *written in C++*. We use the bulk synchronous version as opponent, since it has the best performance among three variants of PowerGraph. We use the CyclopsMT for comparison as PowerGraph is essentially multithreaded.

Table 4 summarizes the comparison between Cyclops and PowerGraph for PageRank algorithms with different datasets and graph partition algorithms. For comparison under hash-based graph partition, the similar hash functions are used to partition graphs based on vertex and edge for Cyclops and PowerGraph accordingly. Though the average number of replicas in Cyclops and PowerGraph are close, each vertex sends 5 messages in each superstep in PowerGraph, of which 3 are used in Gather and Apply phases and the other 2 are used in Scatter phase. Cyclops only requires at most 1 message for vertex in one iteration. Due to the improvement in communication, Cyclops outperforms PowerGraph on Amazon and GWeb datasets. However, for LJournal and Wiki datasets, the performance is affected more by vertex computation, which takes more than 25% and 39% of execution time in Cyclops, respectively. The improvement from communication is not enough to overcome the performance gap between languages.

For heuristic graph partitioning, Cyclops uses Metis algorithm and PowerGraph uses Coordinated Greedy algorithm to partition graphs. We did not use the same partition algorithm as the former tries to minimize edges to cut while the latter tries to minimize vertices to cut. As the average numbers of replicas are still comparable, the results are similar to that of hash-based partition.

7. RELATED WORK

Large-scale graph processing: The emergence of social network and web search have stimulated the development of a number of large-scale graph processing. MapReduce [11] and its relatives [17, 7] have been shown to effectively process web graphs by ranking pages [5] and other graph-related algorithms [25, 35]. Based on MapReduce and its relatives, there have been several systems such as PEGASUS [19], Presto [33], HADI [18] and MadLinq [29] that extend such platforms for graph processing. However, the iterative nature and cross-computation dependence in typical graph algorithms may result in suboptimal performance for large-scale graph processing [22, 21, 12] on such platforms.

Piccolo [26] uses a distributed key-value table abstraction to allow computation on different machines to access shared and mutable states. Unlike Piccolo, the immutable view abstraction provides just read-only access to distributed shared graph views and thus is immune to possible data races and does not have to worry about consistency issues. Trinity [31] in an on-going research project that supports online graph processing, which, however, has to confront users from the consistency models. It also proposes to restrict message passing by buffering and aggregating messages in memory. However, it still requires message passing to access the cached messages. Kineograph [10] aims at online graph processing by using an epoch commit protocol on periodic snapshots, which may be beneficial for extending Cyclops to support incremental graph processing.

Bulk synchronous parallel: Since its first invention by Valiant [32], there have been a number of BSP library implementations, including BSPlib [16] and Greep BSP library [13]. There have also been several extensions to the BSP model, including the hierarchical BSP model [6], which is similar to the hierarchical processing optimization in Cyclops.

Graph Replication and Partition: Parallel BGL [14] distributes graphs by using a property map to store values corresponding to a vertex. It introduces the ghost cell, which allows a vertex to access values through the put/get interfaces to the property map. However, the ghost cell in parallel BGL is write accessible and an application-specific resolver is required to arbitrate and combine messages, which may cause consistency issues and incur burdens to programmers. In contrast, the immutable view provided in Cyclops allows read-only replication of vertices and is with clearer semantics and potentially better scalability due to elimination of frequent coherence messages. Pujol et al. [28] describe an online partition scheme that tries to minimize replicas of vertices through joint partition and replication. This can benefit Cyclops from reducing requires replicas without comprising load balancing. The Surfer [9] captures the network unevenness in cloud environments during graph partitioning by accounting both the machine topology graph and data graph. Such an integrated partitioning may improve performance of Cyclops running in cloud environments.

8. CONCLUSION AND FUTURE WORK

This paper identified issues with existing graph computation models and presented Cyclops, a synchronous vertex-oriented graph processing system that is easy to program and provide efficient vertex computation, yet with significantly less messages than prior systems. We showed that Cyclops performed comparably with PowerGraph despite the language difference due to less messages. A release of Cyclops is available at: `http://ipads.se.sjtu.edu.cn/projects/cyclops/cyclops-snapshot-0.1.tar.gz`

Cyclops currently has no support for topology mutation of graph yet, as its baseline system (i.e., Hama) does not have such a feature yet. We plan to add such support in our future work. Further, we currently evaluated Cyclops only in a small-scale in-house cluster. We plan to study its performance in a larger scale cluster using Amazon EC2-like cloud platforms. In addition, Some features of efficient software and hardware, such as zero-copy protocol in Ibis [3] and RDMA in Infiniband, can also benefit Cyclops. We will consider them in future work.

9. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments. This work is supported in part Doctoral Fund of Ministry of Education of China (Grant No. 20130073120040), the Program

for New Century Excellent Talents in University of Ministry of Education of China, Shanghai Science and Technology Development Funds (No. 12QA1401700), a foundation for the Author of National Excellent Doctoral Dissertation of PR China, China National Natural Science Foundation (No. 61003002) and Singapore NRF (CREATE E2S2).

10. REFERENCES

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

[2] Apache. The Apache Hama Project. http://hama.apache.org/.

[3] H. E. Bal, J. Maassen, R. V. van Nieuwpoort, N. Drost, R. Kemp, T. van Kessel, N. Palmer, G. Wrzesinska, T. Kielmann, K. van Reeuwijk, et al. Real-world distributed computer with ibis. *Computer*, 43(8):54–62, 2010.

[4] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and distributed computation: numerical methods*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[5] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *WWW*, pages 107–117, 1998.

[6] H. Cha and D. Lee. H-bsp: A hierarchical bsp computation model. *J. Supercomput.*, 18(2):179–200, 2001.

[7] R. Chaiken, B. Jenkins, P. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: easy and efficient parallel processing of massive data sets. *VLDB Endowment*, 1(2):1265–1276, 2008.

[8] K. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM TOCS*, 3(1):63–75, 1985.

[9] R. Chen, M. Yang, X. Weng, B. Choi, B. He, and X. Li. Improving large graph processing on partitioned graphs in the cloud. In *ACM SOCC*, 2012.

[10] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen. Kineograph: taking the pulse of a fast-changing and connected world. In *EuroSys*, pages 85–98, 2012.

[11] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[12] J. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, 2012.

[13] M. Goudreau, K. Lang, S. Rao, T. Suel, and T. Tsantilas. Portable and efficient parallel computing using the bsp model. *IEEE Trans. Computers*, 48(7):670–689, 1999.

[14] D. Gregor and A. Lumsdaine. The Parallel BGL: A generic library for distributed graph computations. *Parallel Object-Oriented Scientific Computing (POOSC)*, 2005.

[15] H. Haselgrove. Wikipedia page-to-page link database. http://haselgrove.id.au/wikipedia.htm, 2010.

[16] J. Hill, B. McColl, D. Stefanescu, M. Goudreau, K. Lang, S. Rao, T. Suel, T. Tsantilas, and R. Bisseling. BSPlib: The BSP programming library. *Parallel Computing*, 24(14):1947–1980, 1998.

[17] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, pages 59–72, 2007.

[18] U. Kang, C. Tsourakakis, A. Appel, C. Faloutsos, and J. Leskovec. HADI: Fast diameter estimation and mining in massive graphs with Hadoop. *ACM TKDD*, 5:8:1–8:24, 2011.

[19] U. Kang, C. E. Tsourakakis, and C. Faloutsos. PEGASUS: A Peta-Scale Graph Mining System Implementation and Observations. In *ICDM*, pages 229–238, 2009.

[20] G. Karypis and V. Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. In *Int. Conf. Supercomputing*, 1996.

[21] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: Large-scale graph computation on just a PC. In *OSDI*, 2012.

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

[23] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A New Parallel Framework for Machine Learning. In *Conf. on Uncertainty in Artificial Intelligence*, Catalina Island, California, 2010.

[24] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146, 2010.

[25] B. Panda, J. Herbach, S. Basu, and R. Bayardo. PLANET: massively parallel learning of tree ensembles with MapReduce. *VLDB Endowment*, 2(2):1426–1437, 2009.

[26] R. Power and J. Li. Piccolo: building fast, distributed programs with partitioned tables. In *OSDI*, pages 1–14, 2010.

[27] S. N. A. Project. Stanford large network dataset collection. http://snap.stanford.edu/data/.

[28] J. Pujol, V. Erramilli, G. Siganos, X. Yang, N. Laoutaris, P. Chhabra, and P. Rodriguez. The little engine (s) that could: scaling online social networks. In *ACM SIGCOMM*, pages 375–386, 2010.

[29] Z. Qian, X. Chen, N. Kang, M. Chen, Y. Yu, T. Moscibroda, and Z. Zhang. MadLINQ: large-scale distributed matrix computation for the cloud. In *EuroSys*, pages 197–210, 2012.

[30] S. Salihoglu and J. Widom. GPS: A Graph Processing System. http://infolab.stanford.edu/gps/, 2012.

[31] B. Shao, H. Wang, and Y. Li. The trinity graph engine. Technical Report 161291, Microsoft Research, 2012.

[32] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, Aug. 1990.

[33] S. Venkataraman, E. Bodzsar, I. Roy, A. AuYoung, and R. S. Schreiber. Presto: distributed machine learning and graph processing with sparse matrices. In *Proc. EuroSys*, pages 197–210. ACM, 2013.

[34] C. Wilson, B. Boe, A. Sala, K. P. Puttaswamy, and B. Y. Zhao. User interactions in social networks and their implications. In *EuroSys*, pages 205–218, 2009.

[35] J. Ye, J. Chow, J. Chen, and Z. Zheng. Stochastic gradient boosted distributed decision trees. In *ACM CIKM*, pages 2061–2064, 2009.

[36] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan. Large-scale parallel collaborative filtering for the netflix prize. In *Int. Conf. on Algorithmic Aspects in Information and Management*, pages 337–348, 2008.