# MOCgraph: Scalable Distributed Graph Processing Using Message Online Computing

Chang Zhou[†] Jun Gao[†] Binbin Sun[*] Jeffrey Xu Yu[‡]
[†] Key Laboratory of High Confidence Software Technologies, EECS, Peking University
[*] Central Software Institute, Huawei Technologies Co., Ltd
[‡] Department of Systems Engineering & Engineering Management, Chinese University of Hong Kong
{zhouchang,gaojun}@pku.edu.cn, sunbinbin@huawei.com, yu@se.cuhk.edu.hk

## ABSTRACT

Existing distributed graph processing frameworks, *e.g.,* Pregel, Giraph, GPS and GraphLab, mainly exploit main memory to support flexible graph operations for efficiency. Due to the complexity of graph analytics, huge memory space is required especially for those graph analytics that spawn large intermediate results. Existing frameworks may terminate abnormally or degrade performance seriously when the memory is exhausted or the external storage has to be used.

In this paper, we propose MOCgraph, a scalable distributed graph processing framework to reduce the memory footprint and improve the scalability, based on message online computing. MOCgraph consumes incoming messages in a streaming manner, so as to handle larger graphs or more complex analytics with the same memory capacity. MOCgraph also exploits message online computing with external storage to provide an efficient out-of-core support. We implement MOCgraph on top of Apache Giraph, and test it against several representative graph algorithms on large graph datasets. Experiments illustrate that MOCgraph is efficient and memory-saving, especially for graph analytics with large intermediate results.

## 1. INTRODUCTION

Recently, several in-memory distributed graph processing frameworks, *e.g.,* Pregel [7], GraphLab [23], Giraph [1], GPS [18], are proposed to tackle with general graph analytics. They adopt a vertex-centric computational model, which is very friendly for users to write and debug their parallel graph algorithms. These frameworks exploit main memory to avoid costly disk random accesses, which can be incurred by the poor spatial locality in common graph operations.

Although these frameworks have been widely used for their simplicity and efficiency, the memory limitation issue lurks beneath their facades. Parallel graph algorithms are usually complex, which may spawn large volumes of intermediate results. These intermediate data, including messages transmitted across the network, values and states associated with each vertex, can easily exceed the memory limit, leading to performance degradation or even program crash. For example, we evaluate the minimum memory requirement for running Triangle Counting on Twitter dataset (48M vertices and 2.8B undirected edges, disk storage of 28GB) in Giraph 1.0 and GraphLab 2.2, two representative distributed graph processing frameworks. We find that Giraph and GraphLab need 370GB and 800GB respectively to run the case, indicating 13x and 28x memory explosions with respect to the graph size. As the memory resources become exhausted, the performance of Giraph degrades seriously due to the growing Java GC overheads or its inefficient out-of-core execution, while GraphLab simply crashes because of out-of-memory.

It's necessary to reduce the memory footprint in these graph processing frameworks, which potentially enables them to support larger graphs or more complex graph analytics within the same memory capacity, thus improves the scalability.

Currently, there are several high level techniques to deal with the memory footprint issue, all of which are far from ideal. First, Combiner [7] can be used to reduce the number of buffered messages. However, it cannot access to the vertex value when combining messages, which restricts its expressiveness. In addition, there are still lots of combined messages to be preserved in each superstep. Second, out-of-core execution can be a supplementation to the in-memory computing flow. Among the popular distributed graph processing frameworks, Giraph [1] provides an out-of-core execution support since version 1.0. However, the performance is affected by its costly sort-merge processing style for the out-of-core messages. Another technique used to reduce cross-machine messages is the advanced graph partitioning strategies [11, 19, 8, 20]. Though they can be applied to the existing frameworks, extra work needs to be done either by preprocessing or setting up a global lookup table.

Engineering efforts have also been devoted to improve the memory usage. For example, Facebook uses Giraph to undertake its graph mining tasks with trillion edges [4]. According to their report, *"Reducing memory use was a big factor in enabling the ability to load and send messages to 1 trillion edges"*. They reduce the memory footprint by modifying the underlying in-memory data structures such as byte array representation for vertices and edges, native java unsafe serialization, just name a few.

In this paper, we also seek opportunities to reduce the memory footprint in distributed graph processing frameworks. We observe that, messages in Pregel-like systems can be computed as soon as they arrive for a large number of graph analytics. We call this computing style as *message online computing*. In such a way, the space for buffering messages can be greatly saved, which reduces the potential disk I/Os that may be incurred by the memory shortage.

There are several challenges in designing and implementing the


*Proceedings of the VLDB Endowment,* Vol. 8, No. 4

message online computing (MOC) model. Firstly, MOC-model should have sufficient expressive power to support varieties of graph analytics, while reducing the memory footprint. Secondly, message online computing acts like an asynchronous operation, but we still need to develop an efficient synchronous execution engine to exploit its advantages. Thirdly, we have to provide an efficient out-of-core support unified with the MOC-model, in order to deal with larger graphs or more complex graph analytics.

In this paper, we propose MOCgraph, a scalable distributed graph processing framework using MOC-model, to reduce the memory footprint and improve the scalability. We list several contributions of this paper as follows:

- We introduce a MOC-model to process large graphs. MOC-model can save memory by digesting the incoming messages on the fly. The model has a sufficient expressive power, and is more space-efficient than Combiner which is commonly used in memory reduction.
- We design MOCgraph, a distributed graph processing framework that runs on top of MOC-model. MOCgraph supports both synchronous and asynchronous executions, and has an efficient out-of-core engine. Its fault-tolerance mechanism enables quick restart as no messages are dumped. Optimization strategies for the out-of-core engine such as edge separation and hot-aware re-partitioning are proposed to further reduce disk I/Os.
- We provide succinct and unified programming APIs for both in-memory and out-of-core executions.
- We implement MOCgraph on top of Apache Giraph. Extensive experiments illustrate that MOCgraph is efficient and memory-saving. For space-costly graph analytics with limited memory resources, MOCgraph can run 20x faster than Giraph. MOCgraph can also achieve a comparable time cost with GraphLab while having a nearly 8x memory reduction in some cases.

The rest of the paper is organized as follows. We first review the pregel-like systems on the internal structure and data flow in section 2, then introduce our framework based on the message online computing in section 3. In section 4, two optimizations are proposed to reduce the buffered messages in the out-of-core execution. In section 5, we illustrate the programming APIs. Section 6 evaluates our framework in terms of running time and memory usage. Finally, we review the related works in section 7 and then conclude the paper in section 8.

## 2. REVIEW OF PREGEL-LIKE SYSTEMS

In this section, we give a brief introduction to the internal data flow for a typical superstep in Apache Giraph [1], to shed light on the memory usage issues in existing pregel-like systems. We choose Giraph as it's a well-known open source implementation of Pregel [7] and is widely used for its convenience to cope with the Hadoop ecosystem [4]. Besides, it provides the out-of-core execution, which is needed when the memory is in short.

Giraph inherits the computational model in Pregel [7], which is built on Bulk Synchronous Parrallel (BSP) [22] model. It adopts a vetex-centric view for easy programming, so that users only need to "think as a vertex" and write a *compute()* function for each vertex to execute.

Figure 1 illustrates the data flow in a typical superstep within a worker of Giraph. In Giraph, graphs are partitioned and assigned

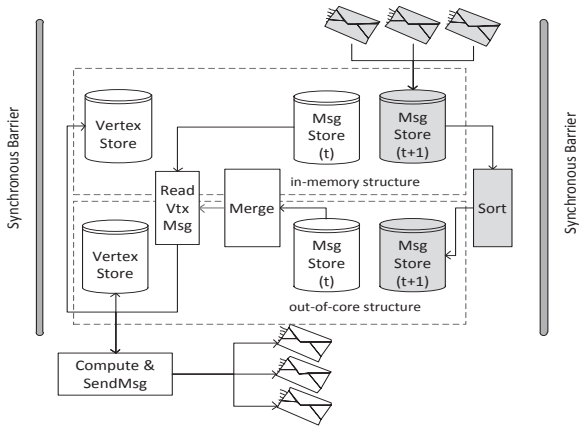



**Figure 1: Data Flow in Original Giraph at Superstep t**

to multiple workers. A vertex store in each worker manages several graph partitions assigned to it. Each vertex $v$ is preserved in exactly one graph partition, where the framework can access to its information by an index with the $v.id$ as the key. This information includes the vertex value and the out-going edges. Messages sent to $v$ are organized separately in the message store.

Once a superstep starts, each vertex calls the *compute()* function, consumes all of its messages stored by the last superstep and updates its value. During the *compute()*, it can send messages to its neighbors based on the updated value. Note that these messages can only be used in the next superstep, which is determined by the computational model in Pregel. At the end of each superstep, the coordination phase ensures that all the messages have been received and preserved completely. Thus, in superstep $t$, Giraph has to maintain two message stores, $MsgStore_t$ and $MsgStore_{t+1}$, as illustrated in Figure 1. The former keeps the messages received in superstep $t-1$, and these messages are fed to the vertex computation in superstep $t$. The latter buffers the incoming messages generated in superstep $t$, which implies that they will not be consumed until superstep $t+1$.

Giraph also supports out-of-core execution, by storing partial vertices and messages on disk, marked as "out-of-core structure" in Figure 1. Messages flushed out will be sorted first, so that all messages for a vertex can be concatenated by merging the top elements of the message files, which prevents message overflow for a given partition.

Now we discuss the inefficiency for Giraph in terms of its memory usage and out-of-core support. Giraph has to keep all the incoming messages in memory and do nothing with them until the next superstep, which may cause the memory shortage. Restricted by the computational model, other pregel-like systems also have the similar problem. For the out-of-core execution in Giraph, operations like sort and merge are also expensive, which can degrade the performance.

A common strategy to reduce the space cost for the incoming messages is to use Combiner [7]. A Combiner may be implemented either at the sender-side or the receiver-side. The sender-side Combiner can reduce the number of messages that transmitted over the network, but it does not work well as expected in distributed graph processing frameworks, because of the poor spatial locality among the destination vertices, as reported in [18]. In fact, neither Giraph [1] nor GPS [18] has implemented the sender-side Combiner. Receiver-side Combiner, though, releases some pressures brought to memory, it suffers from three major problems. First, Combiner

is not applicable for many graph analytics such as Semi-Cluster and Triangle Counting, in which the vertex values have to be accessed or updated during the message combining. Second, Combiner still needs to cache one combined message for each vertex, which is also remarkable when the message is of large size. Third, it's not natural to express Combiner in a vertex centric model. Users have to write two separate functions, *combine* and *compute*, which is cumbersome for some complex graph analytics.

# 3. ONLINE COMPUTING FRAMEWORK

In this section, we introduce MOCgraph, a distributed graph processing framework based on message online computing, aiming to achieve efficient memory usage and high scalability.

We first propose a message online computing model, then compare it with the existing vertex-centric models in terms of expressiveness and space overheads in section 3.1. Then we discuss the synchronous and asynchronous execution modes of MOC-model in section 3.2. Later we illustrate our implementation of MOC-model in section 3.3. The fault-tolerance mechanism is discussed in section 3.4.

## 3.1 Message Online Computing Model

We first review the vertex-centric computational models of Pregel and Pregel with Combiner [7]. Then we introduce the MOC-model and compare their expressive powers and memory usages. These vertex-centric models differ from each other by the way they express the vertex updating and the message processing.

We explain the notations as follows. $D(v)^i$ represents the vertex value of $v$ at superstep $i$. $M^i_{in}$ and $M^i_{out}$ stand for the incoming and outgoing messages for $v$ at superstep $i$, respectively. $m^i_k(1 \leq k \leq |M^i_{in}|)$ is the $k$-th message in the arriving stream of $M^i_{in}$.

**Pregel-model.** Vertex program described by Pregel can be formalized as

$$M^i_{in} \leftarrow \bigodot_{m^i_k \in M^i_{in}} m^i_k$$
$$(D(v)^{i+1}, M^{i+1}_{out}) \leftarrow compute(D(v)^i, M^i_{in})$$

where operator $\odot$ concatenates the messages of $m^i_k$ to compose $M^i_{in}$. Compute function updates the vertex value according to $M^i_{in}$ and generates outgoing messages for $v$. We call this computational model as Pregel-model. Graph analytics that can be written by Pregel-model are then categorized in Pregel-class.

In Pregel-model, for incoming message $m^i_k$ at superstep $i$, $\odot$ happens in the same superstep. *Compute* fuction, however, is called at superstep $i+1$ to digest these messages. Clearly, the data carried across two successive supersteps contains all the $M_{in}$ and $D(v)$, which can exhaust the memory resources.

**Combiner-model.** Vertex program described by Pregel with Combiner can be formalized as

$$m^i_{in\_combined} \leftarrow \bigoplus_{m^i_k \in M^i_{in}} m^i_k$$
$$(D(v)^{i+1}, M^{i+1}_{out}) \leftarrow compute(D(v)^i, m^i_{in\_combined})$$

where $\oplus$ is a *combine operator* which has to be commutative and associative. We call this computational model as Combiner-model. Analytics that can be written by this model are in Combiner-class.

Instead of simply concatenating, Combiner-model combines the incoming messages all along the way. For analysis simplicity, in this paper, we require $\oplus$ to have an effect of reducing memory space after two messages are combined, that is, we disallow $\oplus$ to degenerate into $\odot$ used by Pregel-model. Notice that, although the $\oplus$ operator can reduce the number of incoming messages to be 1 per vertex, it cannot get access to $D(v)^i$ during its computation, which restricts its expressiveness.

Another popular vertex-centric model is GAS-model [8], which can exploit a sum function to combine the gathered messages. GAS-model can emulate both Pregel-model and Combiner-model as suggested in [8], which has exactly the same memory consumption when emulating them, if the extra space cost for vertex replications is not considered. Therefore we only compare our model to Pregel-model and Combiner-model in the rest of the paper.

Now we formally define the MOC-model as follows.

DEFINITION 1. ***Message Online Computing(MOC)-model.*** *In MOC-model, vertex program can be written as*

$$M^i_{out} \leftarrow sendMessages(D(v)^i_k)$$
$$D(v)^i_k \leftarrow onlineCompute(D(v)^i_{k-1}, m^i_k)$$
$$D(v)^i_0 \leftarrow D(v)^{i-1}_{final}$$

*MOC-model updates the value of $v$ with the message stream, forming an update sequence of $D(v)^i_k(1 \leq k \leq |M^i_{in}|)$ during superstep $i$. $v$ has an initial value of $D(v)^i_0$ at the beginning of superstep $i$, where $D(v)^i_0$ is the final state of $D(v)^{i-1}_{final}$ at superstep $i-1$. $v$ is scheduled to send messages only once, based on $D(v)^i_k$ that is available at the time of sending. Messages here are required to be commutative, that is, the final state of the vertex value at each superstep is regardless of the message updating order. The applicable class for MOC-model is then called* ***MOC-class****.*

Notice that, MOC-model produces and processes messages in the same superstep, thus having no messages carried across supersteps, which lowers space overhead.

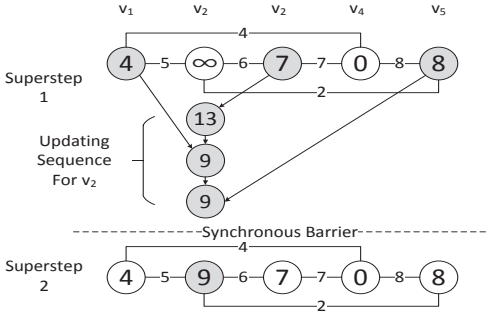



**Figure 2: SSSP using Online Computing**

EXAMPLE 1. ***SSSP Using Message Online Computing.*** *Figure 2 shows an example for Single Source Shortest Path(SSSP) using message online computing. Circles that lie in the same column represent the update sequence of the same vertex, according to a top-down timeline. The value in a vertex is its current shortest distance to $v_4$, and the shaded vertices are active to send message. The raw distance between two vertices is marked on the edge. We can see that, at superstep 1, vertex $v_2$ receives messages from $v_3$, $v_1$, $v_5$ successively. Instead of keeping the messages, $v_2$ online computes these messages one by one, and updates its own value from $\infty$ to 13, 9, 9 accordingly. At superstep 2, $v_2$ has no message to compute, thus only needs to send messages based on the value of 9, which is updated during the last superstep. Clearly, in message online computing model, no message needs to be saved if the memory can hold all the vertices.*

Now we compare these models in terms of expressiveness, memory usage and succinctness. First we discuss the *effective expressive power* of the three computational models and compare their applicable classes. Notice that, if we allow the operators degenerating to concatenation operator $\odot$, these models are all expressive equivalent. Thus, we only compare their effective applicable-classes, in which their functionalities to reduce the space cost should take effects. Clearly, Pregel-model is the most expressive model among the three, as it has no constrains when dealing with the vertex updating and the message sending. Correspondingly, Pregel-class is the widest class. MOC-model can emulate the Combiner-model, while the latter cannot get access to the vertex value when combining the messages, which indicates that MOC-class contains Combiner-class. In fact, for any vertex program in Combiner-class, we can extend $D(v)$ to be $(D(v), m_{empty})$, and combine the message stream only with $m_{empty}$ in *onlineCompute* operator. Thus, after all messages have been processed, the vertex value of $(D(v), m_{combined})$ is prepared. Then we can update the vertex value in *sendMessage* function and send message based on it, which has exactly the same computing behaviour with the Combiner program. However, Graph analytics like Triangle Counting and Semi-Clustering are in MOC-class but outside of Combiner-class, in which messages cannot be combined without reading or writing the vertex value.

Second, Combiner-model is more space-efficient than Pregel-model, as it carries only one message per vertex across the superstep. We then argue that MOC-model is more space-efficient than Combiner-model. In the worst case, we can achieve the same space cost by emulating the Combiner-model as discussed above. However, for analytics whose messages can be directly melt into the vertex value like PageRank, Shortest Path, Connected Components and Landmark Construction, MOC-model ideally has no need to store any message, which makes it more space-efficient than Combiner.

Besides, MOC-model is more succinct than Combiner-model. In Combiner-model, users need to write *compute* and *combine* function with different views on vertex updating (vertex-messages) and message combining (message-message). In contrast, there's only one such function (vertex-message) to be implemented with MOC-model.

## 3.2 Synchronous and Asynchronous Computation with MOC-model

The MOC-model described in Definition 1 is asynchronous, since it sends messages based on the latest vertex value available at the sending time. Asynchronous computation has been demonstrated to speedup the convergence of many iterative graph analytics, as studied in [2, 6]. On the other hand, it is also highly needed for MOC-model to support synchronous execution, which preserves the determinism during the computation. By **determinism**, we mean that any vertex value at the end of each superstep can be determined before runtime. Determinism can be helpful for convergence designing and program debugging, and it can also contribute to the predictable performance.

In order to support synchronous evaluation of MOC-model, we restrict the input of *sendMessages* operator at superstep $i$ to be the initial state of $D(v)^i_0$, instead of any available $D(v)^i_k$ at the time of message sending. As the final vertex state $D(v)^i_{final}$ is determined regardless of the updating order, we can predict the vertex value at the end of each superstep, which preserves determinism.

To implement the synchronous MOC-model, we have two options to deal with a message $m$ that arrives for an unscheduled vertex $v$, namely, wait until $v$ is scheduled to send its messages or update on a copy of $v$. In the former option, we still have to buffer a large number of such messages, which may take too much space and go against our goals. Thus we follow the latter idea in this paper and update on a copied value of $v$, if $v$ has not been scheduled. That is, there are optionally two values attached to $v$ in each superstep, namely the initial value *v.oldValue* and the latest value *v.value* respectively.

**Algorithm 1** Graph-Level Scheduling

```
1: function SEND
2:     for each vertex v do
3:         sendMessage(v);
4:     end for
5: end function
6: function RECEIVE(msgs)
7:     for each msg msg in msgs do
8:         v ← locateVertex(msg.targetVid);
9:         onlineCompute(msg, v);
10:    end for
11: end function
12: function SETUP
13:    if SynchronousEngine then
14:        wait for global value switch synchronous barrier
15:        for each vertex v do
16:            v.oldValue ← v.value;
17:            v.value ← createNewVersion(v.oldValue);
18:        end for
19:    end if
20: end function
```

**Algorithm 2** Vertex-Level Computation

```
1: function SENDMESSAGE(v)
2:     if SynchronousEngine then
3:         v.sendMessage(v.oldValue);
4:     else
5:         v.sendMessage(v.value);
6:     end if
7: end function
8: function ONLINECOMPUTE(msg,v)
9:     v.onlineCompute(msg, v.value);
10: end function
```

The graph-level and the vertex-level scheduling algorithm can be depicted in Algorithm 1 and 2 respectively. In graph-level scheduling, *send* function schedules each vertex to send message *exactly once* in a superstep. Simultaneously, *receive* function accepts incoming messages and online computes them. In the synchronous mode, extra setup is called for each vertex to switch the two values of *v.value* and *v.oldValue* between two successive supersteps, surrounded by an extra global synchronous barrier, which indeed can be avoided as will be discussed in section 3.3. In vertex-level computation with the synchronous engine specified, we will send message based on the vertex value at the beginning of that superstep, namely *v.oldValue* in Line 3. Any incoming message will update on the most recent value of *v*, namely *v.value*, no matter whether synchronous or asynchronous execution is specified, as shown in line 9.

From Algorithm 1 and 2, it is easy to know that once the synchronous mode is specified, MOCgraph follows the synchronous MOC-model described earlier. As for asynchronous mode, it enables vertices to send messages based on the latest vertex value, which may contribute to a faster convergence.

## 3.3 MOCgraph: an Implementation of MOC-model

Now we present MOCgraph, an implementation of MOC-model on top of Apache Giraph [1]. Figure 3 illustrates the data storage and the message online computing data flow within a typical superstep in a worker. We then explain the major differences between MOCgraph and Giraph.

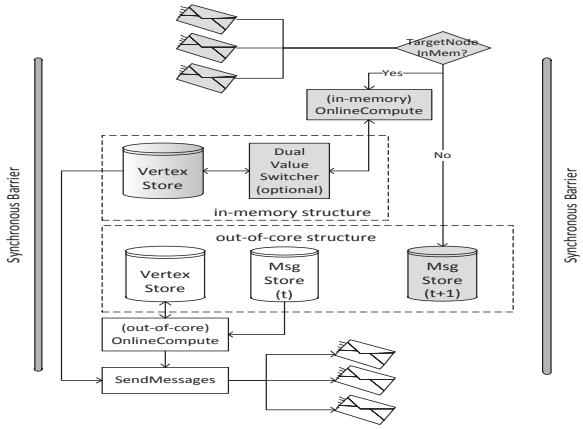



**Figure 3: Data Flow in MOCgraph at Superstep t**

**Data Storage.** There are two major types of data that need to be preserved, the graph topology data and the intermediate data during the computation, including the attached vertex values and messages. Like Giraph, we organize the vertices and their messages separately in *vertex store* and *message store*.

*Vertex Store.* The vertex store is almost the same with Giraph, except that in MOCgraph, an extra copy of vertex value may be also attached temporally to the vertex when the synchronous execution is specified.

*Message Store.* Different from Giraph, the message store in MOCgraph only serves the out-of-core engine and preserves those messages whose target vertices are out-of-core. The message store contains several message files, each of which will be fed to the corresponding vertex partition for the computation in the next superstep. In addition, we only need to append the message stream to the message file rather than organize the messages according to their target vertex ids, which can avoid expensive sort or merge. This append-only behavior makes it efficient to deal with massive messages.

**Data Flow.** We demonstrate the data flow in MOCgraph in a view of working threads. In the workflow of MOCgraph, there are three major components, namely *in-memory online computing*, *out-of-core online computing* and *dual version switcher*. These components are located in two different typs of threads, namely message-receiving thread (shaded) and message-sending thread (white), as shown in Figure 3. Message-receiving thread is passive to process every message that comes for the worker, while message-sending thread is self-motivated to schedule each active vertex to send its messages, exactly once per superstep. Accordingly, in the scheduling algorithm shown in Algorithm 1, *send* is in the message-sending thread and *receive* is in message-receiving thread. Both threads start once a superstep begins.

*In-memory Online Computing.* The shaded *OnlineCompute* component in Figure 3 lies in the message-receiving thread. As the vertex store is accessed by both the sending thread and the receiving thread, concurrency control mechanism is needed. Compared with the granularity of vertex-level locking [8], the partition-granularity locking we use incurs less contention. Once a batch of messages for a vertex partition $p$ arrives, we try to lock $p$ in the local memory. As soon as the write lock is attained, the in-memory online computing will consume these messages and update the corresponding vertex values. If $p$ is out-of-core, we only *append* these messages to the message file for $p$.

*Out-of-core Online Computing.* The out-of-core *OnlineCompute* component resides in the message-sending thread. Once the thread starts, it will traverse all the vertex partitions to send messages. For any vertex partition $p$, the message-sending thread first loads $p$ into memory if it's laid on disk. Before $p$ can start sending messages, it has to consume all its belonging messages left on disk to update the vertices in $p$. For each message $m$ of $p$ on disk, the message-sending thread finds its target vertex $v$ in $p$, online computes $m$ and updates $v$. After all messages of $p$ have been processed, $p$ is scheduled to send messages for each of its vertices based on the updated value.

*Dual Version Switcher.* In order to implement the synchronous MOC-model in MOCgraph, we can simply follow the scheduling algorithm illustrated in Algorithm 1 and 2. However, this synchronous scheduling is not efficient in terms of both space and time cost, in which unnecessary vertex values may be created and an extra global synchronization is needed for the value switch between two successive supersteps. MOCgraph provides a *dual version switcher* as a proxy for visiting the vertex values as shown in Figure 3, to support synchronous execution while reducing such overheads. Specifically, we create a new value of $v$ if any message arrives before $v$ sends messages, and delete the old version as soon as possible. In addition, we use the current superstep number to switch the values logically, which avoids the synchronization overheads for physical switch. The whole algorithm is given in algorithm 3.

**Algorithm 3** Dual Version Switch

```
1:  // Optimized code for synchronous execution in Algorithm 2
2:  function SYNCHRONOUSONLINECOMPUTE(msg, v)
3:      newValue ← v.getNewValue();
4:      if newValue == null then
5:          oldValue ← v.getOldValue();
6:          newValue ← createNewValue(oldValue);
7:      end if
8:      v.onlineCompute(msg, newValue);
9:  end function
10: function SYNCHRONOUSSENDMESSAGE(v)
11:     oldValue ← v.getOldValue();
12:     newValue ← v.getNewValue();
13:     v.sendMessages(oldValue);
14:     if newValue == null then
15:         newValue ← createNewValue(oldValue);
16:     end if
17:     oldValue ← null;
18: end function
19: // Logical Vertex Value Switch
20: function GETNEWVALUE
21:     return getSuperstep()%2==0?v.value0:v.value1;
22: end function
```

Messages received for $v$ always update on *newValue*, and the first message is responsible for initializing the *newValue* based on *oldValue* if *newValue* is not ready, as shown in lines 2-9. In synchronous mode, each active vertex is scheduled to send messages based on its old value, exactly one time per superstep. After that, it

will create *newValue* based on *oldValue* if *newValue* is not attached, then the *oldValue* is deleted. The way of value creation is depicted by the *createNewValue()* function, which by default switches between the two values.

We avoid the extra global synchronization as well as the physical value switch, by using the current superstep number as a flag to tell which physical value is indeed the *newValue*. Lines 20-22 in algorithm 3 illustrate the way to retrieve a logical new value. The retrieval of a logical old value is similar hence we omit it.

Now we analyze that dual version switch can preserve determinism using inductive reasoning. The initialized condition is trivial. Suppose at superstep $i$, we can determine any vertex value at the end of the superstep before runtime. Then at superstep $i+1$, for any vertex $v$, we send messages based on the old value of $D(v)_0^{i+1}$, which is determined at the end of superstep $i$ and ensured by the logical switch. Thus $M_{in}^{i+1}$ can be determined before runtime. Lines 5-7, 14-16 ensure that any $m_k^{i+1}$ in $M_{in}^{i+1}$ will update on $D(v)_0^{i+1}$ successively. As messages in MOC-model are commutative, we can determine the final value of $v$ before runtime. Therefore, dual version switch preserves the determinism.

**Comparison with Giraph.** In-memory MOCgraph is much more space-efficient than Giraph. Asynchronous engine in MOCgraph can eliminate the space cost incurred by two message stores with an expected size of $|M|S_m$, where $|M|$ is the number of messages and $S_m$ is the average size per message. For synchronous execution, we need another space of $0.5|V|S_v$ to hold the new values expectedly using the dual version switch approach. The space reduction then becomes to $|M|S_m - 0.5|V|S_v$, where $|V|$ is the number of vertices and $S_v$ is the average size of a vertex value. As for message intensive graph analytics, $|M|$ has the same order of magnitude as $|E|$ which is considered to be much bigger than $|V|$. Thus, the space reduction is still notable even we use the synchronous engine.

We also have advantages if the vertices cannot fully reside in memory, even though we have to preserve messages targeted at out-of-core vertices as Giraph does. First, MOCgraph can reduce the disk I/Os because part of messages can be consumed in memory. Second, MOCgraph avoids costly I/O operations such as sort or merge during the message dumping and loading. Last but not least, MOCGraph can support graph analytics on nature graphs well. In large natural graphs, there are high fan-in vertices with a lot of messages, which may occupy huge amount of memory resources if we load them all in memory before computing as Giraph does. Fortunately, MOCgraph processes the out-of-core messages as streams, and therefore doesn't suffer from that issue.

## 3.4 Fault-Tolerance

MOCgraph uses the periodic checkpointing strategy to achieve fault-tolerance, with much fewer messages saved compared to other Pregel-like systems. Take Giraph as an example, it has to dump all of the graph topology, vertex values and incoming messages at the end of a superstep, to be used for the superstep restart. These overheads are determined by the Pregel-model, in which vertex values have to be updated based on the messages in the previous superstep.

In contrast, in MOCgraph, the synchronous barrier ensures that vertex values have been updated by the incoming messages in the current superstep. This makes a superstep stateless with respect to the messages, so that if we purely use the in-memory execution, no messages are required to be checkpointed. When the out-of-core execution is enabled, in-memory messages also need not be saved since they've done their jobs in updating the vertices, while the out-of-core messages are naturally checkpoint files. In this way, less data is recorded, leading to a faster restart.

# 4. OPTIMIZATIONS

In this section, we introduce two optimizations for the out-of-core engine that can further reduce the number of disk I/Os. In Section 4.1, we propose an edge separation strategy to let more vertices be in memory, allowing more messages to be online-computed in memory. Section 4.2 provides a partition rearrangement and replacement strategy within local machines.

## 4.1 Edge Separation

We observe that *compute()* function in many graph analytics does not need the information on the out-going edges to update the vertex value, where edges are only used to send messages. These tasks include Pagerank, Connected Components, Landmark Index Construction, BFS, Shortest Path, just name a few. For these graph analytics, we can safely separate the edges from a vertex partition, and sacrifice the memory space of these edges to let more vertex partitions be in memory, such that each incoming message has much more probabilities to be computed on the fly. We call this strategy edge separation as the vertex partition no longer contains the edge information.

Now we analyze the percentage of I/O reduction using edge separation strategy. We use $S_m$, $S_v$, $S_e$ to represent the average size per message, per vertex value and per edge value, respectively, and $|M|$, $|V|$, $|E|$ represent the number of messages, vertices and edges, accordingly. Let $C$ be the memory capacity, $\lambda_{basic}$ and $\lambda_{es}$ be the fraction of the vertex partitions that can reside in memory, for the basic strategy and the edge separation strategy respectively. We have,

$$C = \lambda_{basic}(|V|S_v + |E|S_e) = \lambda_{es}|V|S_v \tag{1}$$

with constrains of

$$0 < \lambda_{basic} < \lambda_{es} \leq 1 \tag{2}$$

Assume messages are uniformly distributed among vertex partitions, the I/O cost for these two strategies in one superstep will be as follows,

$$\begin{aligned} cost_{basic} &= 2(1-\lambda_{basic}) \cdot (|V|S_v + |E|S_e + |M|S_m) \\ cost_{es} &= 2(1-\lambda_{es}) \cdot (|V|S_v + |M|S_m) + 2|E|S_e \end{aligned} \tag{3}$$

in which the out-of-core vertex partition associated with the messages sent to it needs to be read and written once. As for edge separation, though it requires to load and offload edges of in-memory vertex partitions, $\lambda_{es}$ can be much larger than $\lambda_{basic}$ , and we can calculate the I/O reduction by:

$$\begin{aligned} reduction &= 1 - \frac{cost_{se}}{cost_{basic}} \\ &= \frac{|M|S_m}{|V|S_v(|V|S_v + |M|S_m + |E|S_e)} \\ &\cdot \frac{|E|S_e}{[(|V|S_v + |E|S_e)/C - 1]} \end{aligned} \tag{4}$$

As can be inferred by equation (1)(2)(4), we have

$$\begin{aligned} C &\leq |V|S_v \\ 0 < reduction &\leq \frac{|M|S_m}{|V|S_v + |M|S_m + |E|S_e} \end{aligned} \tag{5}$$

From equation (4)(5), we can see that the edge separation strategy can always spawn less I/O compared to the basic method and gains the maximum reduction when $C = |V|S_v$, that is, the memory is just enough to hold all the edge-separated vertex partitions. For example, in PageRank, $S_v = S_m = S_e$, and $|M| = |E| \gg$

$|V|$ for many graphs, then we can get a nearly $50\%$ I/O reduction theoretically by using edge separation.

## 4.2 Hot-Aware Re-Partitioning

In MOCgraph, vertex partitions have to be swapped onto disk when memory is insufficient, so we need a well-suited partition replacement policy for MOCgraph, to reduce the total number of disk I/Os.

First, we need to decide whether we should load partition $p$ into memory or offload the messages onto disk when the messages come for an out-of-core vertex partition $p$. We choose the latter strategy and only allow the message-sending thread to do the vertex partition replacement, because the messages arrive continuously in an unpredictable manner and it's not worth frequently swapping a vertex partition in and out.

Then we choose a proper replacement strategy for the message-sending thread. According to the actions we take, the victim partition swapped out cannot consume messages belonging to it until it's reloaded into memory. If we assume the messages arrive uniformly for each partition, any partition replacement strategy will have the same effects. However, when the messages are not uniformly distributed among the partitions, we can always keep those partitions that tend to receive more messages to be in memory. Therefore, the number of message I/Os can be lowered.

Based on these observations, we propose a hot-aware partitioning method, to differentiate the expected number of messages of each partition, providing opportunities for a more suitable replacement strategy.

DEFINITION 2. ***Hot/Cold Partition.*** *A vertex partition p is hot if its heat is in the top-k list of the local partitions, where the heat of p is regarded as the expected number of messages that it will receive, and k is the maximum number of partitions that can reside in memory. Other partitions are then the cold partitions. In this paper, we measure the heat by the total number of edges in p.*

A widely used graph partitioning method is hash partitioning, which generates roughly equal-sized partitions with respect to both vertex number and edge number. Though it's good at eliminating data skew across different workers as well as space-efficient, we cannot benefit directly from it. Our purpose is to differentiate the partition heats within each local worker, while maintaining load balance across workers. Thus, we can create opportunities for devising an I/O efficient partition replacement strategy. Now we introduce the hot-aware re-partitioning as follows.

DEFINITION 3. ***Hot-Aware Re-Partitioning.*** *Let $P = \bigcup_{i=1}^{n} p_i$ be the partition sets assigned to the local worker, hot-aware re-partition is to reorganize P into $P' = \bigcup_{i=1}^{n} p'_i$, such that*

$$P' = \arg\max_{P'}\Big(\sum_{p'_h \in P'_{hot}} heat(p'_h) - \sum_{p'_c \in P'_{cold}} heat(p'_c)\Big)$$

$$s.t. \sum_{p'_h \in P'_{hot}} size(p'_h) = totalMemForVtx$$

*where $P'_{hot}$ and $P'_{cold}$ represent the hot/cold partition set in $P'$ respectively, and the maximal size of memory allocated for vertices is represented by totalMemForVtx.*

It's easy to have an exact solution of this equation, *e.g.,* sorting vertices in all local partitions by degree and reassign them to new partitions accordingly. However, it's worth doing so especially for cases that memory is not enough to hold all the vertices. In this paper, we use a streaming strategy to reorganize the local partitions. We process the vertices as streams within a buffer of size $k * p$, where $p$ is the number of the generated partitions and $k$ is the parameter to control the partition effects. We read $k * p$ vertices from local partitions, sort them according to their degree, and assign the vertex to the $i$-th partition if it has the $i$-th largest degree. Experiments in section 6.3 show that a small $k$ of *100* or *1000* works just fine.

The hot-aware re-partitioning is a reorganization strategy of the local vertex partitions. It can be applied to any existing graph partition technique, as it does not break the vertex-worker ownership. We only need to maintain a local lookup table after the repartition, which is space-efficient. The graph partitioning method we use in this paper is hash partitioning, mainly for its power of eliminating data skew across workers.

Notice that, hot-aware re-partition strategy alone may be less effective for reducing the total disk I/Os, because hot partitions may be also in large size with more edges, which means more partitions have to be left out-of-core. However, we can use the hot-aware re-partitioning strategy together with the edge separation technique introduced earlier, because the size of each hot partition is roughly the same as that of each cold partition. Thus we can decrease the number of messages failed to be online computed in memory, by letting hot partitions in memory.

EXAMPLE 2. *Figure 4 shows an example for the hot-aware re-partitioning. Vertex 1,3,5,7 are sent to worker 1 by hash partitioning, then partitions 1,3 are reorganized to form new ones with various partition heat. Suppose we only have room for one partition, we prefer partition 1 to be in memory since it has more edges, so that there will be potentially more messages consumed on the fly, thus reduce the disk I/Os.*

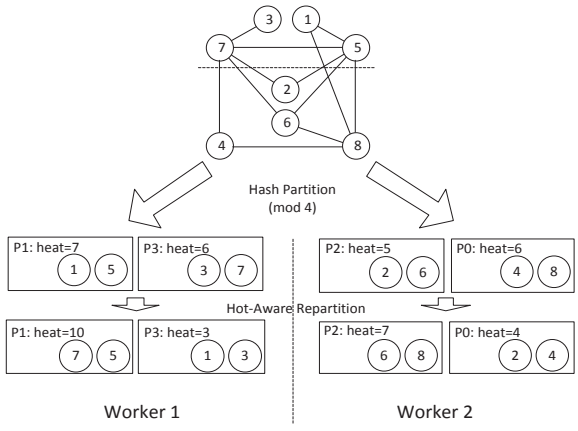



**Figure 4: Hot-Aware Partitioning**

Now we depict the whole picture of vertex partition replacement strategy in MOCgraph. We reorganize the vertex partitions within each worker to differentiate the edge counts among local partitions, and keep all the hot partitions in memory before the first superstep. We leave one partition space to exchange with the cold partitions left on disk. Before the message-sending thread starts online computing and sending messages of an out-of-core partition, it will move out the cold partition that previously swapped in. By doing this, we can have more messages online computed in memory, thus reduce the number of messages to be preserved.

Next, we illustrate how the hot-aware re-partitioning can help lower the disk I/Os. Let $\mu_{cold} \leq 1$ represent the ratio of average

number of edges in cold partitions to that in all partitions, and other notations have the same meaning as in section 4.1. We have

$$cost_{es+hot} = 2(1-\lambda_{es}) \cdot (|V|S_v + \mu_{cold}|M|S_m) + 2|E|S_e$$
$$= cost_{es} - 2(1-\lambda_{es})(1-\mu_{cold})|M|S_m \quad (6)$$

The more notable effects of the re-partitioning strategy, the smaller $\mu_{cold}$ will be, therefore the more disk I/Os can be reduced, as can be seen in the equation (6).

# 5. PROGRAMMING APIS

In this section, we provide the programming APIs used in MOCgraph, and give an example to demonstrate their usages.

We illustrate the major part of APIs here in Figure 5, in order to tell the differences with Giraph. We split the *compute()* function into two parts, *onlineCompute()* and *sendMessages()*, with totally different responsibilities.

```
// user applied computing function
void compute(Iterator<M> msgs);
// combine the two messages to form one
// which is from Combiner class
void combine(M oldMsg, M newMsg);
// vertex value accessor
V getValue();
void setValue(V value);
// get the current superstep number
long getSuperstep()

// online compute with a single message
void onlineCompute(M msg, V newValue, int
superstepNum);
// send messages based on the appropriate
// value provided by the framework
void sendMessages(V value);
// optional, user can overwrite this for
// synchronous execution
V createNewVersion(V oldValue);
```

**Figure 5: API Changes in MOCgraph compared to Giraph**

The function of *onlineCompute()* describes only about how to compute a message and update the vertex value, while *sendMessages()* is responsible only for sending messages based on the current value, which is called only once during a superstep. The framework will choose the correct vertex value as the input of *sendMessages()* and *onlineCompute()* automatically, according to the execution mode specified by users. Thus, we hide the original value accessor in Giraph from user, to avoid obfuscation. Combiner is also no more needed and we simply remove it.

The *createNewVersion()* function tells MOCgraph about how to generate a new value based on the old one, when the synchronous execution is specified. The default way is to switch the old value to the new, which can be adopted by many graph analytics. Users can override this function to meet their own requirements.

EXAMPLE 3. ***SSSP Example.*** *We list our SSSP implementation using Giraph 1.0 and MOCgraph in Figure 6 and Figure 7, respectively. We can see that the original compute() is divided into onlineCompute() and sendMessages(). The onlineCompute() function computes a message and updates the current distance. In sendMessages(), a vertex will send its changed distance to its neighbors if it's waked up. For asynchronous execution, the distance sent by each vertex is the latest version available at the sending time, while in synchronous mode, it's the initial value at each superstep. Users are not aware of the existence of the two versions, which are auto-provided by the framework.*

```
public void compute(Iterable<DoubleWritable>
    messages) throws IOException {
  double minDist = isSource() ? 0d :
      Double.MAX_VALUE;
  for (DoubleWritable message : messages) {
    minDist = Math.min(minDist,
        message.get());
  }
  if (minDist < getValue().get()) {
    setValue(new DoubleWritable(minDist));
    for (Edge<LongWritable, DoubleWritable>
        edge : getEdges()) {
      double distance = minDist +
          edge.getValue().get();
      sendMessage(edge.getTargetVertexId(),
          new DoubleWritable(distance));
    }
  }
  voteToHalt();
}
```

**Figure 6: Original SSSP in Giraph**

```
public void onlineCompute(DoubleWritable
    message, DoubleWritable curDst, int
    superstep) throws IOException {
  double dist = messge.get();
  if(dist < curDst.get()){
    curDst.set(dist);
    //wake up the vertex explicitly
    //so that it can be scheduled to send
        messages.
    wakeUp();
  }
}
//only awaken vertices are scheduled to send
    messages
public void sendMessages(DoubleWritable dst){
  double curDst = dst.get();
  for (Edge<LongWritable, DoubleWritable>
      edge : getEdges()) {
    double distance = curDst +
        edge.getValue().get();
    sendMessage(edge.getTargetVertexId(),
    new DoubleWritable(distance));
  }
  voteTohalt();
}
```

**Figure 7: SSSP For Message Online Computing**

# 6. EXPERIMENT

In this section, we conduct extensive experiments to evaluate the performance of MOCgraph, in terms of the task execution time and the memory usage.

## 6.1 Experimental Setup

**Environment.** All experiments are evaluated on a 23-node cluster. Each node has two 2.60GHz AMD Opteron 4180 processors with

48GB of RAM and 2TB hard disk. SUSE Linux Enterprise Server 11 and Java 1.7 with 64-bit server JVM are installed on these nodes. All these nodes are connected to the same 1GB network switch. We use Giraph version 1.0 and Hadoop version 0.20.3, with the HDFS block size of 64M.

**Dataset.** Three real-world graph datasets are used in our experiments. *uk-2007-05* is a time-aware graph generated by combining 12 monthly snapshot of the .uk domain [15]. *twitter-2010* is a follower-following topology in the social network Twitter [12]. *livejournal* is a friendship network of an online community site LiveJournal. We also have some synthetic random graphs whose numbers of edge range from 1 billion to 5 billion. We convert these graphs into undirected ones since some of our applications need an undirected graph as input. Some statistics about these graphs are summarized in Table 1. Each edge is assigned a weight randomized between 1 and 100.

| DataSet | # Nodes | # UndirectedEdges |
|---|---|---|
| *livejournal* | 4,847,571 | 68,993,773 |
| *twitter-2010* | 41,652,230 | 2,405,026,390 |
| *uk-2007-05* | 105,896,555 | 6,625,316,807 |
| *ranNb* | 0.05*N billion | N billion |

**Table 1: Statistics of Graph DataSets**

**Graph Analytics.** We test our framework on several representative graph analytics, with different features on the size of vertex value and the size of message value, as listed in Table 2.

| Graph Analytic Task | Size of Vertex Value | Size of Message |
|---|---|---|
| *PageRank(PR)* | small | small |
| *Connected Components(CC)* | small | small |
| *Reachability(RC)* | small | small |
| *Single Source Shortest Path(SSSP)* | small | small |
| *Triangle Counting(TC)* | small | large |
| *Semi-Cluster(SC)* | large | large |
| *Landmark Index Construction(LM)* | large | large |

**Table 2: Types of Graph Analytics**

**Competitors.** We compare MOCgraph with Giraph of version 1.0 and GraphLab of version 2.2. We directly use the CC, PR, TC, SSSP examples provided in the toolkits of GraphLab 2.2, and implement LM by our own.

**Experiments Design.** We first illustrate the benefits brought by the asynchronous execution of MOCgraph in section 6.2. We further evaluate the effects of the proposed optimization strategies in section 6.3, namely edge separation and hot-aware re-partition. Then we compare MOCgraph with Giraph, Giraph Combiner and their out-of-core supports varying the memory capacity in section 6.3. In section 6.5, we compare MOCgraph with GraphLab separately since GraphLab has no out-of-core implementation.

## 6.2 Synchronous vs. Asynchronous Execution in MOCgraph

In order to have a clear sight about how asynchronous execution in MOCgraph can help speedup the tasks, we run three different graph analytics on *Twitter* with sufficient memories using both synchronous and asynchronous execution mode, as illustrated in Figure 8. We will show more comprehensive results when the memory capacity is taken into account in section 6.3.

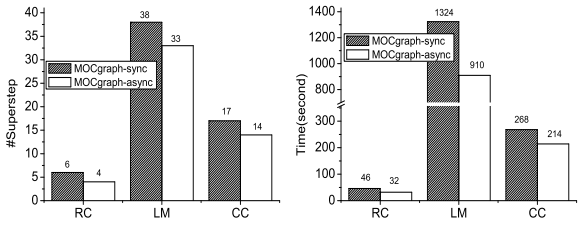


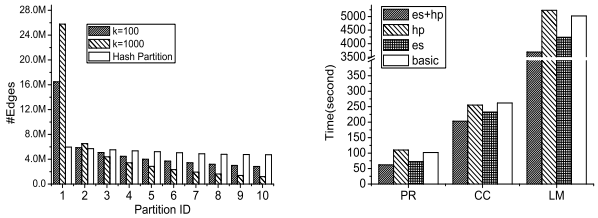

**Figure 8: Superstep and Time Cost for both sync and async mode: we choose Twitter as the dataset, and set the number of workers 23, 23, 46 for Reachability, Connected Components and Landmark, respectively. The number of landmarks is set to 25. All cases are running with sufficient memory.**

Figure 8 shows that asynchronous execution in MOCgraph can accelerate the convergence of many applications such as Landmark Index Construction, Connected Components and Reachability, gaining notable reductions of supersteps, which contributes to the time speedup. In fact, in asynchronous execution of MOCgraph, vertices send messages based on their most recent values, which may be updated by some incoming messages in the current superstep. These experiments demonstrate that although the asynchronous computation in MOCgraph still reside in supersteps, it can also speedup the convergence of many graph analytics as many other asynchronous systems [23, 8] do.

## 6.3 Effects for Optimization Strategies

Next, we test the performance for the optimization strategies proposed in section 4, namely edge separation(*es*) and hot-aware re-partition(*hp*), against the basic MOCgraph. As the optimization works for out-of-core execution, we only assign 1GB memory for each worker, and run three graph analytics with different strategies.

Figure 9(a) shows the edge number within each partition in a typical worker after the hot-aware re-partitioning on *Twitter*. $k$ is a user-specified parameter to control the size of the sort buffer, as mentioned in section 4. We can see that even with a small *k=100*, MOCgraph can still differentiate the partition heat remarkably, which provides an opportunity to further reduce the number of disk I/Os.




(a) Partition Heat (b) Time Cost

**Figure 9: Effects for edge separation and hot-aware re-partition**

As can be seen in Figure 9(b), edge separation strategy(*es*) is effective in all the cases, which gains 30%, 10%, 16% time speedup for PR, CC, and LM, respectively. Hot-aware re-partition strategy(*hp*) alone is less effective or even degrade the performance, because hot partitions are also in large size, as explained in section 4.2. However, the combination of *es* and *hp* works well for all these tasks. Take PageRank as an example, *es* works well because

it reduces the large amount of disk I/Os as illustrated in section 4.1. For *hp*, hot partitions also have more edges, which take up more spaces, so that the increased number of online computed messages are trivial. For *hp+es*, it makes more vertices be in memory hence decreases the number of disk I/Os.

## 6.4 Comparison with Giraph and Combiner

We compare MOCgraph with Giraph, Giraph Combiner, as well as their out-of-core supports.

Parameter tuning is not easy for out-of-core execution in Giraph. Given a limited memory capacity, we have to manually specify the out-of-core options in Giraph. We need to specify whether the vertex partitions and the messages should be outside of memory separately. Then we may need to provide two other parameters: *maxPartitionsInMemory* and *maxMessagesInMemory*. The former indicates how many vertex partitions can reside in memory, while the latter limits the number of messages that can fit in memory.

We try our best to do the parameter tunings in Giraph's out-of-core engine using following strategies: our first choice is to use in-memory messages and out-of-core vertices with Combiner if it supports. If the task fails, we tend to have as many as partitions to be in memory while adjusting the number of *maxMessagesInMemory*. The other parameters are set by default. We only report the best performance of the out-of-core Giraph as long as we can run it since there are too many failures during our tests.

Figure 10 illustrates the running time (graph loading time not included) of MOCgraph and its competitors, on graph analytics mentioned in table 2, varying the memory capacity per worker.

*MOCgraph vs. Giraph.* We can see that in Figure 10, MOCgraph can be an order of magnitude faster than Giraph with limited memory on message-intensive analytics, including PR, TC, LM, and SC. The performance of Giraph degrades seriously as the memory capacity becomes less, because of the inefficient out-of-core execution. For TC on *Twitter* in Figure 10(a), with memory capacity of 2g*46, MOCgraph can finish within 40 minutes, while out-of-core Giraph fails after 10 hours, which leads to a 15x speedup. Similarly, MOCgraph spends 25 minutes to finish LM of 100 landmarks on *LiveJournal* with 1g*23 memory, while it costs Giraph for more than 500 minutes to do the same job.

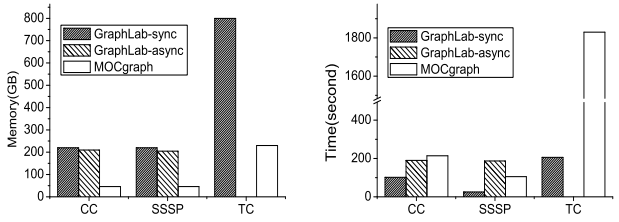
*MOCgraph vs. Combiner.* MOCgraph is more expressive than Combiner. For example, Combiner cannot be applied to TC and SC in Figure 10(a) and 10(b). For graph analytics with large intermediate results that can be expressed by Combiner, MOCgraph-async can run much faster than Combiner. For LM in Figure 10(b), MOCgraph-async achieves a 2x speedup than Combiner for 3g*46 memory capacity and a 5x speedup for 2.5g*46 on *LiveJournal*. When there's only 2g*46 memory, the running time of Combiner soars incidently, and the speedup for MOCgraph-async comes to 15x. This is because the memory is in short to hold all combined messages while we cannot make it run in the out-of-core mode, so that the excessive garbage collection overheads degrade the performance of Combiner seriously. Although MOCgraph-sync has the similar time cost when the memory is sufficient, it can also achieve a 9x speedup to Combiner when the memory is limited to 2g*46, since it can exploit message online computing using the out-of-core engine of MOCgraph.

For analytics whose message and vertex values are both small, MOCgraph can still outperform Combiner, but the speedup drops. For CC shown in Figure 10(c), MOCgraph-async gains the speedup of 1.5x for both *Twitter* and *UK* datasets, while the MOCgraph-sync achieves very similar performance with Combiner. For PR in Figure 10(d), we only evaluate the MOCgraph-sync to make this comparison on the same output. Similar with the case explained for Figure 10(b), MOCgraph-sync can also outperform Combiner in PR, when the memory cannot hold all the combined messages.

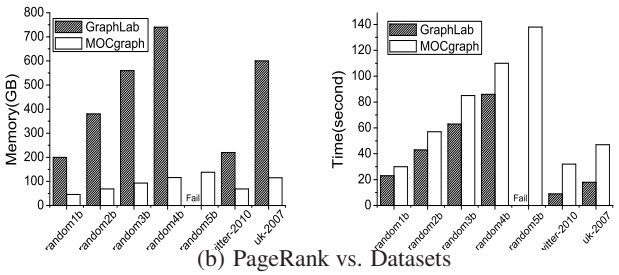
*MOCgraph-sync vs. MOCgraph-async.* In these cases, we can see that MOCgraph-async runs faster than MOCgraph-sync. The reasons are twofolds. i) MOCgraph-async can accelerate the convergence therefore reduce the total number of supersteps as indicated in section 6.2. ii) MOCgraph-sync has extra overheads to create copies of the vertex values.

## 6.5 Comparison with GraphLab

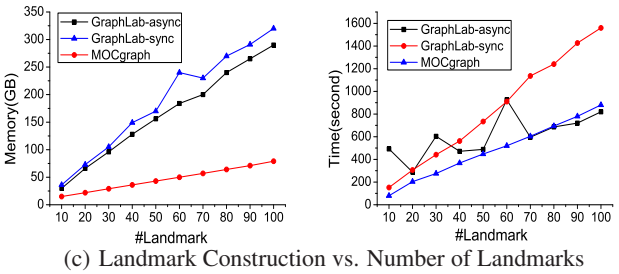
We conducted experiments against GraphLab, as illustrated in Figure 11. Note that, GraphLab currently does not have an out-of-core support, so we run MOCgraph all in memory to have a fair comparison.




(a) Memory Usage and Time Cost vs. Analytics



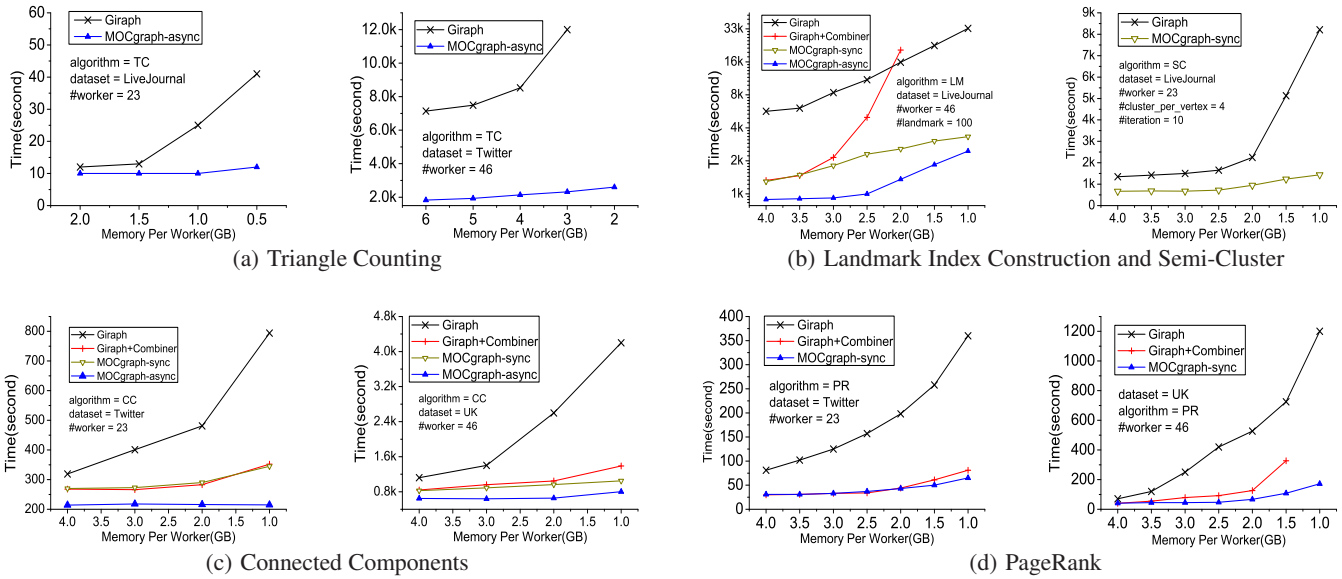

(b) PageRank vs. Datasets




(c) Landmark Construction vs. Number of Landmarks

**Figure 11: Comparison with GraphLab**

We first evaluate graph analytics of CC, SSSP, TC on *Twitter* in Figure 11(a). We can see that GraphLab-sync is faster than MOCgraph at the price of huge memory consumptions. This is because GraphLab exploits a vertex-replication strategy to reduce inter-machine messages. Though it can lower the time cost, the memory burden increases. We also notice that GraphLab-async does not perform well as expected. For example, it runs nearly 2x and 7x slower than its async engine in CC and SSSP respectively. This may due to its heavy lock contention, communication overheads in distributed locking and the lack of message batching, which has also been discussed in [9]. In contrast, MOCgraph-async usually runs faster than MOCgraph-sync, as can be seen in Figure 10. In fact, the asynchronous execution in MOCgraph also resides in supersteps, which allows asynchronous vertex updating to




(a) Triangle Counting

(b) Landmark Index Construction and Semi-Cluster

(c) Connected Components

(d) PageRank

**Figure 10: Memory Usage and Speedups: Lines marked as *Giraph* represent tasks running with Giraph. *MOCgraph* and *Giraph* can run both in-memory and out-of-core. *Giraph+Combiner* stands for tasks running with Giraph and Combiner, which can run all in memory, or with message partitions all in memory but vertex partitions out of core.**

speedup convergence while reducing lock contentions by exploiting message batching and partition-granularity locking.

Figure 11(b) compares the performance in terms of space/time cost for running PageRank on varies of datasets. We run each case for 5 iterations and report the average time. As can be seen that, memory consumptions of GraphLab is about 5x larger than MOCgraph, and GraphLab even fails to allocate memory with *random5b* dataset. This is mainly due to the additional space for storing local vertex replications. For natural graphs, this replication strategy is effective in reducing the time cost, and it's 2-3x faster than MOCgraph on PageRank. However, the performance gap is narrowed for random graphs, which are also commonly seen in bi-directional social networks.

For applications with large intermediate results, MOCgraph can achieve a similar or better performance than GraphLab. In Figure 11(c), we increase the number of landmarks and show the time/space cost. The results illustrate that time/space cost increases linearly as the landmark number increases for both GraphLab and MOCgraph. GraphLab is still much more memory-consuming than MOCgraph, and GraphLab-sync is even slower than MOCgraph for running LM. In fact, even if a tiny part of the vertex value (a distance to a landmark) changes, GraphLab has to synchronize the whole value (the distance table) among all the replicas, which is very expensive. In contrast, MOCgraph allows users to send only the affected parts, which saves a lot of time.

In addition, we should stress that the straightforward comparison of Giraph and GraphLab is not the purpose of this paper, due to the different implementation details between them, like programming languages (*e.g.,* GraphLab using C++ and Giraph using Java) and communication layer implementations (MPI vs. Netty).

## 6.6 Summary

To sum up, we have the following observations. i) Asynchronous execution in MOCgraph can speedup the convergence for many iterative graph analytics, which contributes to the time speedup. ii) Message online computing is memory-saving, and the proposed optimization strategies can reduce the amount of disk I/Os, leading to time reduction. iii) Compared with Giraph and its Combiner, MOCgraph can achieve a 20x speedup on applications with large intermediate results (Triangle Counting and Landmark Index Construction), when the memory resources are limited. iv) Although GraphLab is usually faster than MOCgraph, MOCgraph can still gain a significant memory reduction. In addition, MOCgraph can achieve a similar or better performance than GraphLab for some analytics with large intermediate results.

Note that, in our implementation, we neither modified the core in-memory data structures nor did any eager memory resource management. The performance gain may indicate the benefits of the MOC-model and its data flow design, which can be also applied to other existing distributed graph processing frameworks such as GPS [18] and GraphLab [23].

# 7. RELATED WORKS

In this section, we review the related works on graph processing frameworks.

**Distributed Graph Processing System.** Recently, a bunch of distributed graph processing frameworks have emerged. MapReduce [5] is a general purpose framework for processing big data. However, it's not suitable to process large graphs due to its high iteration costs, too many disk I/Os and complex representation for graph algorithms. Pregel [7] introduces an all-in-memory vertex-centric distributed graph processing framework. It adopts the BSP [22] model and provides friendly user APIs. Apache Giraph [1], an open source implementation of Pregel that runs on top of Hadoop, is used by the Graph Search Service in Facebook [4], which can process graphs with trillion edges. GPS [18] introduces the master computation and optimizations like dynamic partitioning as well as large adjacency lists partitioning.

GraphLab [23] and its latest version PowerGraph [8] abstract the distributed graph operations as Gather, Apply and Scatter(GAS).

PowerGraph requires a sum function to combine the gathered messages. It supports both synchronous and asynchronous executions, and uses a vertex-cut graph partitioning method to handle natural graphs. Though the vertex-cut approach releases the burden of network communications, PowerGraph needs extra memory for storing the vertex mirrors, which is remarkably large especially when the vertex value is of big size.

Giraph++ [21] adopts the idea of "think like a graph" to speedup convergency of applications like Connected Components, by introducing asynchronous computing among local vertices. This strategy works well on graph analytics whose local convergence can help lead to a global convergence. Trinity [3] introduces dedicated memory management, and supports both online and offline graph analytics. However, it's not open-sourced and does not mention its out-of-core support [3].

**Disk-Backed Graph Processing on a Single Machine.** There are also published works of large graph processing on a single machine, such as GraphChi [14], XStream [17], and TurboGraph [10]. These works are disk-backed, because the memory for a single machine is very limited. They improve the performance by reducing the random access to disk or using modern SSD. Although they can achieve comparable performances with distributed systems on their test data, they still face the scalability issue. These systems have inspired us to develop a distributed graph analysis framework with disk support, to improve the scalability for applications with massive intermediate data and for users with limited memories.

**Graph Partitioning.** Large graphs need to be partitioned, stored and processed across multiple machines. There are many works focusing on the problem of graph partitioning [11, 8, 19]. Their partitioning strategies aim at reducing the number of edge-cuts [11] or the number of vertex-cuts [8], to reduce the amount of inter-machine messages. As reported in [13, 16], none of these partitioning methods is always the best. Our hot partition strategy, on the other hand, is a local reorganization strategy, which can be used along with them to reduce the out-of-core I/O overheads.

# 8. CONCLUSION

This paper proposes MOCgraph, a distributed graph processing framework to support scalable graph analytics. MOCgraph adopts the message online computing model, which greatly reduces the memory footprint. MOCgraph supports both asynchronous and synchronous execution, to achieve convergence speedup and determinism respectively. MOCgraph has an streaming-style out-of-core execution, unified with the online computing data flow. Optimizations such as edges separation and hot-aware re-partitioning can be used to further reduce the potential disk I/Os. Experiments indicate that MOCgraph is space-efficient, providing opportunities to support larger graphs or more complex analytics under the same memory capacity. MOCgraph is also time-efficient especially for those graph analytics with large intermediate results.

# ACKNOWLEDGMENTS

This work was supported by NSFC under Grant No. 61272156, Research Grants Council of the Hong Kong SAR, China No. 14209314 and 418512, National High Technology Research and Development Program of China under No. 2012AA011002, and Huawei Innovative Research Planning.

# 9. REFERENCES

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

[2] D. P. Bertsekas and J. N. Tsitsiklis. Parallel and dis-tributed computation: numerical methods. In *Prentice-Hall*, 1989.

[3] B.Shao, H. Wang, and Y.Xiao. Managing and mining billion-scale graphs: Systems and implementations. In *SIGMOD*, 2012.

[4] A. Ching. Scaling apache giraph to a trillion edges. http://tinyurl.com/ktf6d25, 2013.

[5] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[6] A. Frommer and D. B. Szyld. On asynchronous iterations. In *J. Comput. Appl. Math.*, 2000.

[7] G.Malewicz, M.H.Austern, A.J.C.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.

[8] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30, 2012.

[9] M. Han, K. Daudjee, K. Ammar, T. Ozsu, X. Wang, and T. Jin. An experimental comparison of pregel-like graph processing systems. In *VLDB*, 2014.

[10] W.-S. Han, S. Lee, K. Park, J.-H. Lee, M.-S. Kim, J. Kim, and H. Yu. Turbograph: a fast parallel graph engine handling billion-scale graphs in a single pc. In *SIGKDD*, pages 77–85. ACM, 2013.

[11] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998.

[12] K.Haewoon, L.Changhyun, P.Hosung, and M.Sue. What is Twitter, a social network or a news media? In *WWW*, pages 591–600, 2010.

[13] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis. Mizan: a system for dynamic load balancing in large-scale graph processing. In *EuroSys*, pages 169–182, 2013.

[14] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. In *OSDI*, volume 8, pages 31–46, 2012.

[15] P.Boldi, M.Santini, and S.Vigna. A large time-aware graph. *SIGIR Forum*, 42(2):33–38, 2008.

[16] M. Redekopp, Y. Simmhan, and V. K. Prasanna. Optimizations and analysis of bsp graph processing models on public clouds. In *IPDPS*, pages 203–214, 2013.

[17] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: edge-centric graph processing using streaming partitions. In *SOSP*, pages 472–488. ACM, 2013.

[18] S. Salihoglu and J. Widom. Gps: A graph processing system. In *SSDBM*, page 22. ACM, 2013.

[19] I. Stanton and G. Kliot. Streaming graph partitioning for large distributed graphs. In *SIGKDD*, pages 1222–1230. ACM, 2012.

[20] S.Yang, X.Yan, B.Zong, and A.Khan. Towards effective partition management for large graphs. In *SIGMOD*, pages 517–528, 2012.

[21] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson. From" think like a vertex" to" think like a graph. *PVLDB*, 7(3), 2013.

[22] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[23] Y.Low, J.Gonzalez, A.Kyrola, D.Bickson, C.Guestrin, and J.Hellerstein. Distributed graphlab: A framework for machine learning in the cloud. *PVLDB*, 5(8):716–727, 2012.