From "Think Like a Vertex" to "Think Like a Graph"

Yuanyuan Tian†, Andrey Balmin§ [1], Severin Andreas Corsten⋆, Shirish Tatikonda†, John McPherson†
†*IBM Almaden Research Center, USA* {ytian, statiko, jmcphers}@us.ibm.com
§*GraphSQL, USA* andrey@graphsql.com
⋆*IBM Deutschland GmbH, Germany* severin.corsten@de.ibm.com

ABSTRACT

To meet the challenge of processing rapidly growing graph and network data created by modern applications, a number of distributed graph processing systems have emerged, such as Pregel and GraphLab. All these systems divide input graphs into partitions, and employ a "think like a vertex" programming model to support iterative graph computation. This vertex-centric model is easy to program and has been proved useful for many graph algorithms. However, this model hides the partitioning information from the users, thus prevents many algorithm-specific optimizations. This often results in longer execution time due to excessive network messages (e.g. in Pregel) or heavy scheduling overhead to ensure data consistency (e.g. in GraphLab). To address this limitation, we propose a new "think like a graph" programming paradigm. Under this graph-centric model, the partition structure is opened up to the users, and can be utilized so that communication within a partition can bypass the heavy message passing or scheduling machinery. We implemented this model in a new system, called Giraph++, based on Apache Giraph, an open source implementation of Pregel. We explore the applicability of the graph-centric model to three categories of graph algorithms, and demonstrate its flexibility and superior performance, especially on well-partitioned data. For example, on a web graph with 118 million vertices and 855 million edges, the graph-centric version of connected component detection algorithm runs 63X faster and uses 204X fewer network messages than its vertex-centric counterpart.

1. INTRODUCTION

Rapidly growing social networks and other graph datasets require a scalable processing infrastructure. MapReduce [7], despite its popularity for big data computation, is awkward at supporting iterative graph algorithms. As a result, a number of distributed/parallel graph processing systems have been proposed, including Pregel [15], its open source implementation Apache Giraph [1], GraphLab [14], Kineograph [6], Trinity [20], and Grace [23].

The common processing patterns shared among existing distributed/parallel graph processing systems are: (1) they divide input graphs into partitions for parallelization, and (2) they employ a vertex-centric programming model, where users express their algorithms by "thinking like a vertex". Each vertex contains information about itself and all its outgoing edges, and the computation is expressed at the level of a single vertex. In Pregel, a common vertex-centric computation involves receiving messages from other vertices, updating the state of itself and its edges, and sending messages to other vertices. In GraphLab, the computation for a vertex is to read or/and update its own data or data of its neighbors.

This vertex-centric model is very easy to program and has been proved to be useful for many graph algorithms. However, it does not always perform efficiently, because it ignores the vital information about graph partitions. Each graph partition essentially represents a proper subgraph of the original input graph, instead of a collection of unrelated vertices. In the vertex-centric model, a vertex is very short sighted: it only has information about its immediate neighbors, therefore information is propagated through graphs slowly, one hop at a time. As a result, it takes many computation steps to propagate a piece of information from a source to a destination, even if both appear in the same graph partition.

To overcome this limitation of the vertex-centric model, we propose a new **graph-centric** programming paradigm that opens up the partition structure to users and allows information to flow freely inside a partition. We implemented this graph-centric model in a new distributed graph processing system called Giraph++, which is based on Apache Giraph.

To illustrate the flexibility and the associated performance advantages of the graph-centric model, we demonstrate its use in three categories of graph algorithms: graph traversal, random walk, and graph aggregation. Together, they represent a large subset of graph algorithms. These example algorithms show that the graph-centric paradigm facilitates the use of existing well-known sequential graph algorithms as starting points in developing their distributed counterparts, flexibly supports the expression of local asynchronous computation, and naturally translates existing low-level implementations of parallel or distributed algorithms that are partition-aware.

We empirically evaluate the effectiveness of the graph-centric model on our graph algorithm examples. We compare the graph-centric model with the vertex-centric model, as well as with a hybrid model, which keeps the vertex-centric programming API but allows asynchronous computation through system optimization. This hybrid model resembles the approaches GraphLab and Grace take. For fair comparison, we implemented all three models in the same Giraph++ system. In experimental evaluation, we consistently observe substantial performance gains from the graph-centric model especially on well-partitioned data. For example, on a graph with

Proceedings of the VLDB Endowment, Vol. 7, No. 3

[1] This work was done while the author was at IBM Almaden Research Center.

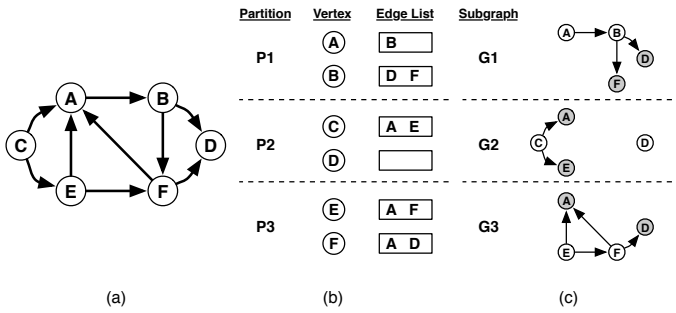


Figure 1: Example graph and graph partitions

118 million vertices and 855 million edges, the graph-centric connected component algorithm ran 63X faster than the vertex-centric implementation and used 204X fewer network messages. This was also 27X faster than the hybrid model, even though it used only 2.3X fewer network messages. These performance gains are due to an algorithm-specific data structure that keeps track of the connected components within a partition and efficiently merges components that turn out to be connected by a path through other partitions. As a result, the graph-centric version needs much fewer messages per iteration and completes in fewer iterations than both the vertex-centric and the hybrid versions.

Note that the proposed graph-centric programming model is not intended to replace the existing vertex-centric model. Both models can be implemented in the same system as we demonstrated in Giraph++. The vertex-centric model has its simplicity. However, the graph-centric model allows lower level access, often needed to implement important algorithm-specific optimizations. At the same time, the graph-centric model still provides sufficiently high level of abstraction and is much easier to use than, for example, MPI[18].

The graph-centric programming model can also be implemented in other graph processing systems. We chose Giraph, due to its popularity in the open source community, and more importantly its ability to handle graph mutation. Graph mutation is a crucial requirement for many graph algorithms, especially for graph aggregation algorithms, such as graph coarsening [12, 11], graph sparsification [19], and graph summarization [22]. Incidentally, to the best of our knowledge, Giraph++ is the first system able to support both asynchronous computation and mutation of graph structures.

The performance of many graph algorithms, especially the ones implemented in the graph-centric model, can significantly benefit from a good graph partitioning strategy that reduces the number of cross-partition edges. Although there has been a lot of work on single-node sequential/parallel graph partitioning algorithms [12, 11, 21], the rapid growth of graph data demands scalable distributed graph partitioning solutions. In this paper, we adapted and extended the algorithm of [11] into a distributed graph partitioning algorithm, which we implemented in the same Giraph++ system, using the graph-centric model.

The remainder of the paper is organized as follows: Section 2 provides a necessary background on Giraph. In Section 3, we introduce the graph-centric programming model, and in Section 4, we exploit the graph-centric model in various graph algorithms. In Section 5, we discuss the hybrid model which is an alternative design to support asynchronous graph computation. Then, the detailed empirical study is provided in Section 6. Section 7 describes the related work. Finally, we conclude in Section 8.

2. GIRAPH/PREGEL OVERVIEW

```
Vertex<I, V, E, M>  //I: vertex ID type, V: vertex
//value type, E: edge value type, M: message type
void compute(); //user defined compute function
long getSuperstep(); //get the current superstep number
void sendMsg(I id, M msg);
void sendMsgToAllEdges(M msg);
void voteToHalt();
boolean isHalted();
int getNumOutEdges(); //get the number of outgoing edges
E getEdgeValue(I targetVertexId);
boolean addEdge(I targetVertexId, E edgeValue);
E removeEdge(I targetVertexId);
Iterator<I> iterator(); //iterator to all neighbors
Iterable<M> getMessages(); //get all messages to it
I getVertexId();
V getVertexValue();
void setVertexValue(V vertexValue);
void write(DataOutput out); //serialization
void readFields(DataInput in); //deserialization
```

Figure 2: Major (not all) functions for Vertex in Giraph.

In this section, we provide an overview of Apache Giraph, which is an open source implementation of Pregel.

Giraph distributes a graph processing job to a set of workers. One of the workers acts as the master to coordinate the remaining slave workers. The set of vertices of a graph is divided into partitions. As shown in Figure 1, each partition contains a set of vertices and all their outgoing edges. Each vertex is uniquely identified by an ID, and a *partitioner* decides which partition a vertex belongs to based on its ID. The partitioner is also used to route messages for a vertex correctly to its partition. The default partitioner is a hash function on the vertex ID. Range partitioner or other customized partitioners can be used as well. The number of partitions is usually greater than the number of workers, to improve load balance.

Giraph employs a vertex-centric model. Each graph vertex is considered an independent computing unit that inherits from the predefined *Vertex* class. Each vertex has a unique ID, a vertex value, a set of outgoing edges (a set of edges in the case of an undirected graph) with an edge value associated with each edge, and a set of messages sent to it. Figure 2 shows the major functions for the *Vertex* class with I as the vertex ID type, V as the vertex value type, E as the edge value type, and M as the message type.

Giraph follows the Bulk Synchronous Parallel (BSP) computation model. A typical Giraph program consists of an input step, where the graph is initialized (e.g., distributing vertices to worker machines), followed by a sequence of iterations, called *supersteps*, which are separated by global synchronization barriers, and finally an output step to write down the results. A vertex carries two states: *active* and *inactive*. In the beginning, all vertices are *active*. A vertex can voluntarily deactivate itself by calling *voteToHalt*() or be passively activated by some incoming messages from other vertices. The overall program terminates if every vertex is inactive. In superstep i, each active vertex can receive messages sent by other vertices in superstep $i-1$, query and update the information of the current vertex and its edges, initiate graph topology mutation, communicate with global aggregation variables, and send messages to other vertices for the next superstep $i+1$. All this computation logic is executed in a user-defined *compute*() function of the *Vertex* class. After all active vertices finish their local *compute*() calls in a superstep, a global synchronization phase allows global data to be aggregated, and messages created by each vertex to be delivered to their destinations.

To reduce the number of messages transmitted and buffered across supersteps, a user can define a *combiner* function if only an aggregate (such as min, max, sum) of messages is required instead of the individual messages.

Algorithm 1: Connected Component Algorithm in Giraph

```
1  COMPUTE()
2      if getSuperstep()==0 then
3          setVertexValue(getVertexID());
4      minValue=min(getMessages(), getVertexValue());
5      if getSuperstep()==0 or minValue<getVertexValue() then
6          setVertexValue(minValue);
7          sendMsgToAllEdges(minValue);
8      voteToHalt();
   // combiner function
9  COMBINE(msgs)
10     return min(msgs);
```

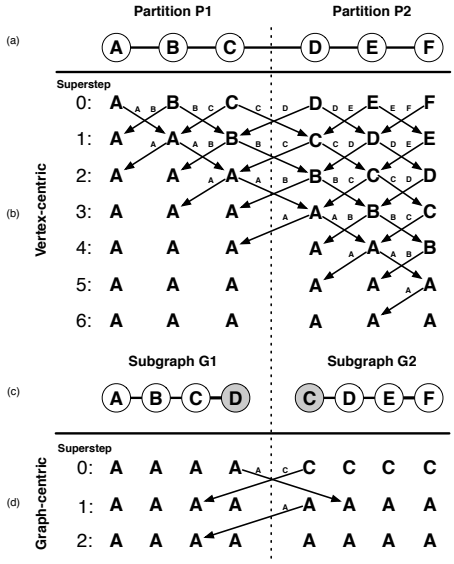


Figure 3: Example execution of connected component algorithms in vertex-centric and graph-centric models

Fault tolerance in Giraph is achieved by periodic checkpointing. Users can specify the frequency of checkpointing (in terms of number of supersteps). During checkpointing (it only happens at the beginning of a superstep), the workers save the state of all vertices including vertex value, edge values, and all incoming messages. Once a failure is detected, the master notifies all workers to enter the recovery mode. Then, in the subsequent superstep, workers reload the full state from the last checkpoint, and proceed with the normal computation.

Example: Connected Component Algorithm in Giraph. Algorithm 1 shows an example of the connected component algorithm for undirected graphs implemented in Giraph. In this algorithm, the vertex value associated with each vertex is its component label. Initially in superstep 0, each vertex uses its own ID as its component label (each vertex is itself a connected component), then propagates the component label to all its neighbors. In subsequent supersteps, each vertex first finds the smallest label from the received messages. If this label is smaller than the vertex's current component label, the vertex modifies its label and propagates the new label to all its neighbors. When the algorithm finishes, the component label for each vertex is the smallest vertex ID in the corresponding connected component. To reduce the number of network messages, a combiner (line 9-10) that computes the min of messages is used for this algorithm. For the example graph in Figure 3(a), Figure 3(b) depicts the vertex labels and the message passing in every superstep for this connected component algorithm.

3. Giraph++ GRAPH-CENTRIC MODEL

In this section, we introduce the new graph-centric programming model. In a nutshell, instead of exposing the view of a single vertex to the programmers, this model opens up the entire subgraph of each partition to be programmed against.

3.1 Internal Vertices and Boundary Vertices

Just like the vertex-centric model, the graph-centric model also divides the set of vertices in the original graph into partitions as depicted in Figure 1(b). Let $G =< V, E >$ denote the original graph with its vertices and edges, and let $P_1 \cup P_2 \cup \ldots \cup P_k = V$ be the k partitions of V, i.e. $P_i \cap P_j = \emptyset, \forall i \neq j$. For each partition P_i, the vertices in P_i, along with vertices they link to, define a subgraph G_i of the original graph. Figure 1(c) shows examples of subgraphs. To be more precise, let V_i denote all the vertices that appear in the subgraph G_i. We define $V_i = P_i \cup \{v | (u, v) \in E \wedge u \in P_i\}$. We say that any vertex $u \in P_i$ is an *internal* vertex of G_i and any vertex $v \in (V_i \backslash P_i)$ is a *boundary* vertex. In Figure 1(c), A and B are the internal vertices of G1, while D and F are its boundary vertices (shown as shaded vertices). Note that a vertex is an internal vertex in exactly one subgraph, which we call the *owner* of the vertex, but it can be a boundary vertex in zero or more subgraphs. For example, in Figure 1(c), A is an internal vertex in G1, and is a boundary vertex in both G2 and G3. G1 is the owner of A.

In the rest of this paper, when discussing graph-centric model, we will refer to subgraphs as partitions when it is not ambiguous to do so.

In the Giraph++ graph-centric model, for each internal vertex in a partition, we have all the information of its vertex value, edge values and incoming messages. But for a boundary vertex in a partition, we only associate a vertex value with it. This vertex value is just a temporary *local* copy. The *primary* copy of the vertex value resides in its owner's corresponding internal vertex. The local copies of vertex values are essentially caches of local computation in different partitions, they have to be propagated to the primary copy through messages.

The distinction between internal vertices and boundary vertices are crucial, as in Giraph++ messages are only sent from boundary vertices to their primary copies. This is because the whole subgraph structure is available in the graph-centric model, information exchange between internal vertices is cheap and immediate. The algorithm can arbitrarily change the state of any internal vertex at any point in time, without a need for a network message or a wait for the next superstep. Boundary vertex values can also be arbitrarily changed, but these changes will have to be propagated to the owners through messages, at the end of the superstep.

3.2 Giraph++ Programming API

As we intend to make the graph-centric model a valuable complement to the existing vertex-centric model, our design principal for Giraph++ is to fully make use of the existing Giraph infrastructure. A program in Giraph++ is still executed in sequence of supersteps, separated by global synchronization barriers. However, in each superstep, the computation is performed on the whole subgraph in a partition.

We utilize the *Vertex* class in Giraph for internal vertices and boundary vertices in Giraph++. However, we turn off functions that are not needed. Among the major functions shown in Figure 2, the retained functions for internal vertices are highlighted in blue and red, whereas only the ones highlighted in red are retained for the boundary vertices. Like in Giraph, each internal vertex of a partition has two states: active or inactive. However, a boundary vertex does not have any state.

```
GraphPartition<I, V, E, M>//I: vertex ID type, V: vertex
//value type, E: edge value type, M: message type
void compute(); //user defined compute function
void allVoteToHalt(); //all vertices vote to halt
long getSuperstep(); //get the current superstep number
void sendMsg(I id, M msg);
boolean containsVertex(I id);
boolean isInternalVertex(I id);
boolean isBoundaryVertex(I id);
Vertex<I, V, E, M> getVertex(I id);
Collection<Vertex<I, V, E, M>> internalVertices();
Collection<Vertex<I, V, E, M>> activeInternalVertices();
Collection<Vertex<I, V, E, M>> boundaryVertices();
Collection<Vertex<I, V, E, M>> allVertices();
void write(DataOutput out); //serialization
void readFields(DataInput in); //deserialization
```

Figure 4: Major functions for GraphPartition.

Algorithm 2: Connected Component Algorithm in Giraph++

```
1  COMPUTE()
2    if getSuperstep()==0 then
3      sequentialCC(); // run a sequential CC algorithm
4      foreach iv IN boundaryVertices() do
5        sendMsg(bv.getVertexId(), bv.getVertexValue());
6    else
7      equiCC=∅;// store equivalent CCs
8      foreach iv IN activeInternalVertices() do
9        minValue=min(iv.getMessages());
10       if minValue<iv.getVertexValue() then
11         equiCC.add(iv.getVertexValue(), minValue);
12     equiCC.consolidate();// get min for equivalent CCs
13     foreach iv IN internalVertices() do
14       changedTo=equiCC.uniqueLabel(iv.getVertexValue());
15       iv.setVertexValue(changedTo);
16     foreach bv IN boundaryVertices() do
17       changedTo=equiCC.uniqueLabel(bv.getVertexValue());
18       if changedTo!=bv.getVertexValue() then
19         bv.setVertexValue(changedTo);
20         sendMsg(bv.getVertexId(), bv.getVertexValue());
21   allVoteToHalt();
```

To support the graph-centric programming model, we introduce a new class called *GraphPartition*. Figure 4 lists the major functions in this class. This class allows users to 1) access all vertices in a graph partition, either internal or boundary, 2) check whether a particular vertex is internal, boundary or neither, 3) send messages to internal vertices of other partitions, and 4) collectively deactivate all internal vertices in this partition. The user defined *compute*() function in the *GraphPartition* class is on the whole subgraph instead of on individual vertex.

Fault Tolerance in Giraph++. Like Giraph, Giraph++ achieves fault tolerance by periodic checkpointing. During each checkpoint, Giraph++ saves all the vertices (with their states), edges (with their states) , and messages in each partition. Furthermore, since users can freely define auxiliary data structures inside the *GraphPartition* class. The checkpointing scheme also calls the *GraphPartition.write()* function to serialize the auxiliary data structures. During failure recovery, besides reading back the vertices, edges, and messages, Giraph++ also deserializes the auxiliary data structures by calling the *GraphPartition.readFields()* function.

Example: Connected Component Algorithm in Giraph++. Algorithm 2 demonstrates how the connected component algorithm is implemented in Giraph++. For the example graph in Figure 3(a), Figure 3(c) and 3(d) depict the subgraphs of its two partitions and the execution of the graph-centric algorithm, respectively.

Sequential connected component algorithms have been well studied in the graph literature. Since the graph-centric programming model exposes the whole subgraph in a partition, an existing sequential algorithm can be utilized to detect the connected components in each graph partition. If a set of vertices belong to the same connected component in a partition, then they also belong to the same connected component in the original graph. After information is exchanged across different partitions, some small connected components will start to merge into a larger connected component.

Exploiting the above property, superstep 0 first runs a sequential connected component algorithm (we use a breath-first-search based algorithm) on the subgraph of each graph partition and then sends the locally computed component label for each boundary vertex to its corresponding owner's internal vertex. For the example in Figure 3(a), superstep 0 finds one connected component in the subgraph G1 and assigns the smallest label A to all its vertices including the boundary vertex D. Similarly, one connected component with label C is detected in G2. Messages with the component labels are then sent to the owners of the boundary vertices. In each of the subsequent supersteps, the algorithm processes all the incoming messages and uses them to find out which component labels actually represent equivalent components (i.e. they will be merged into a larger component) and stores them in a data structure called equiCC. In the above example, vertex D in superstep 1 receives the message A from G1, while its previous component label is C. Thus, pair (A, C) is put into equiCC to indicate that the connected components labeled A and C need to be merged. In equiCC.*consolidate*() function, we use the smallest label as the unique label for the set of all equivalent components. In our example, the new label for the merged components should be A. Then the unique labels are used to update the component labels of all the vertices in the partition. If a boundary vertex's component label is changed, then a message is sent to its owner's corresponding internal vertex. Comparing the two algorithms illustrated in Figure 3(b) and 3(d), the graph-centric algorithm needs substantially fewer messages and supersteps. In superstep 0, all the vertices in P1 already converge to their final labels. It only takes another 2 supersteps for the whole graph to converge. Note that the same combiner function used in Algorithm 1 can also be used in Algorithm 2.

We argue that the graph-centric programming model in Giraph++ is more general and flexible than the vertex-centric model. The graph-centric model can mimic the vertex-centric model by simply iterating through all the active internal vertices and perform vertex-oriented computation. In other words, any algorithm that can be implemented in the vertex-centric model can also be implemented in the graph-centric model. However, the performance of some algorithms can substantially benefit from the graph-centric model. The connected component algorithm in Algorithm 2 is such an example. More examples of the graph-centric model's superior flexibility will be shown in Section 4.

4. EXPLOITING Giraph++

In this section, we explore the application of our proposed Giraph++ graph-centric programming model to three categories of graph algorithms: graph traversal, random walk and graph aggregation. For each category, we pick one representative algorithm to illustrate how it is implemented in the graph-centric paradigm.

4.1 Graph Traversal

Graph traversal represents a category of graph algorithms that need to visit all the vertices of a graph in a certain manner, while checking and updating the values of vertices along the way. These algorithms often involve a search over the graph. Examples include

Algorithm 3: PageRank Algorithm in Giraph

```
1  COMPUTE()
2      if getSuperstep()≤ MAX_ITERATION then
3          delta=0;
4          if getSuperstep()==0 then
5              setVertexValue(0);
6              delta+=0.15;
7          delta+=sum(getMessages());
8          if delta>0 then
9              setVertexValue(getVertexValue()+delta);
10             sendMsgToAllEdges(0.85*delta/getNumOutEdges());
11     voteToHalt();
   // combiner function
12 COMBINE(msgs)
13     return sum(msgs);
```

Algorithm 4: PageRank Algorithm in Giraph++

```
1  COMPUTE()
2      if getSuperstep()>MAX_ITERATION then
3          allVoteToHalt();
4      else
5          if getSuperstep()==0 then
6              foreach v IN allVertices() do
7                  v.getVertexValue().pr=0;
8                  v.getVertexValue().delta=0;
9          foreach iv IN activeInternalVertices() do
10             if getSuperstep()==0 then
11                 iv.getVertexValue().delta+=0.15;
12             iv.getVertexValue().delta+=sum(iv.getMessages());
13             if iv.getVertexValue().delta>0 then
14                 iv.getVertexValue().pr+=iv.getVertexValue().delta;
15                 update=0.85*iv.getVertexValue().delta/iv.getNumOutEdges();
16                 while iv.iterator().hashNext() do
17                     neighbor=getVertex(iv.iterator().next());
18                     neighbor.getVertexValue().delta+=update;
19             iv.getVertexValue().delta=0;
20         foreach bv IN boundaryVertices() do
21             if bv.getVertexValue().delta>0 then
22                 sendMsg(bv.getVertexId(), bv.getVertexValue().delta);
23                 bv.getVertexValue().delta=0;
```

computing shortest distances, connected components, transitive closures, etc. Many such algorithms are well-studied, with sequential implementations readily available in textbooks and online resources.

We just examined the implementation of one such algorithm in the vertex-centric and the graph-centric models, in Algorithms 1 and 2, respectively. In the graph-centric model, we applied an existing sequential algorithm to the local subgraph of each partition in superstep 0, then only propagate messages through the boundary vertices. In the subsequent supersteps, messages to each vertex could result in the value update of multiple vertices, thus requiring fewer supersteps than a corresponding vertex-centric implementation.

4.2 Random Walk

The algorithms of the second category are all based on the random walk model. This category includes algorithms like HITS [13], PageRank [5], and its variations, such as ObjectRank [2]. In this section, we use PageRank as the representative random walk algorithm.

Algorithm 3 shows the pseudo code of the PageRank algorithm (using damping factor 0.85) implemented in the vertex-centric model. This is not the classic PageRank implementation, which iteratively updates the PageRank based on the values from the previous iteration as in $PR_v^i = d \times \sum_{\{u|(u,v)\in E\}} \frac{PR^{i-1}}{|E_u|} + (1-d)$, where $|E_u|$ is the number of outgoing edges of u. Instead, Algorithm 3 follows the accumulative iterative update approach proposed in [25], and incrementally accumulates the intermediate updates to an existing PageRank. It has been proved in [25] that this accumulative update approach converges to the same values as the classic PageRank algorithm. One advantage of this incremental implementation of PageRank is the ability to update increment values asynchronously, which we will leverage in the graph-centric model below.

Algorithm 4 shows the PageRank algorithm implemented in the graph-centric model. This algorithm also follows the accumulative update approach. However, there are two crucial differences: (1) Besides the PageRank score, the value of each vertex contains an extra attribute, $delta$, which caches the intermediate updates received from other vertices in the same partition (line 16-18). (2) Local PageRank computation is *asynchronous*, as it utilizes the partial results of other vertices from the same superstep (line 14). Asynchrony has been demonstrated to accelerate the convergence of iterative computation in many cases [8], including PageRank [14]. Our graph-centric programming paradigm allows the local asynchrony to be naturally expressed in the algorithm. Note that both the vertex-centric and the graph-centric algorithms can benefit from a combiner that computes the sum of all messages (line 12-13 of Algorithm 3).

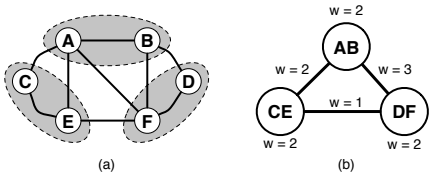


Figure 5: Graph coarsening example

4.3 Graph Aggregation

The third category of graph algorithms, which we call *graph aggregation*, are used to condense a large graph into a structurally similar but smaller graph by collapsing vertices and/or edges. Prominent examples of graph aggregation are graph summarization [22], graph sparsification [19], and graph coarsening [12, 11]. These algorithms are more sophisticated than graph traversal or random walk algorithms. They typically involve mutation of the graph structure by adding and removing vertices or/and edges. As a result, platforms that do not support graph mutation, such as Grace [23] and GraphLab [14], cannot efficiently support these algorithms. We pick graph coarsening as a representative graph aggregation algorithm to implement in the graph-centric paradigm.

4.3.1 Background on Graph Coarsening

Graph coarsening is often used as a first step in a graph partitioning algorithm, to reduce the complexity of the graph. We implemented a modification of the graph coarsening algorithm used in the ParMetis parallel graph partitioning tool [11]. Their coarsening algorithm works on an undirected graph and executes in a number of coarsening rounds to generate smaller and smaller graphs. In each round, the algorithm first finds a maximal matching of the current graph. As shown in Figure 5(a), a matching $M \subseteq E$ is a subset of edges, such that no two edges in M share a common incident vertex. We call a matching M maximal, if it is not a proper subset of another matching of the graph. After finding a maximal matching, the algorithm collapses the incident vertices of each edge in the matching into a *super* vertex. During this coarsening process, the algorithm keeps a weight for each vertex and each edge. Initially,

the weights are all 1. The weight of a super vertex is the sum of the weights of all vertices collapsed into it. An edge between two super vertices is an aggregation of edges between the original vertices, so its weight is the sum of the individual edge weights. Figure 5(b) demonstrates an example coarsened graph of Figure 5(a).

In ParMetis [11], finding a maximal matching is done in a number of phases. In phase i, a processor randomly iterates through its local unmatched vertices. For each such vertex u, it uses a heavy-edge heuristic to match u with another unmatched vertex v if there is any. If v is local, the match is established immediately. Otherwise, a match request is sent to the processor that owns v, conditioned upon the order of u and v: if i is even, a match request is sent only when $u < v$; otherwise, a request is sent only when $u > v$. This ordering constraint is used to avoid conflicts when both incident vertices of an edge try to match to each other in the same communication step. In phase $i + 1$, multiple match requests for a vertex v is resolved by breaking conflicts arbitrarily. If a match request from u is granted, a notification is sent back to u. This matching process finishes when a large fraction of vertices are matched.

4.3.2 Graph Coarsening in Giraph++

The graph coarsening algorithm in ParMetis can be naturally implemented in the graph-centric programming model. The (super) vertex during the coarsening process is represented by the *Vertex* class in Giraph++. When two (super) vertices are collapsed together, we always reuse one of the (super) vertices. In other words, we merge one of the vertex into the other. After a merge, however, we do not delete the vertex that has been merged. We delete all its edges and declare it inactive, but utilize its vertex value to remember which vertex it has been merged to.

Algorithm Overview. The graph coarsening implementation in our graph-centric model follows the similar process as in the parallel ParMetis algorithm: The algorithm executes iteratively in a sequence of coarsening rounds. Each coarsening round consists of m matching phases followed by a collapsing phase. Each of the matching phases is completed by 2 supersteps and the collapsing phase corresponds to a single superstep. We empirically observed that $m = 4$ is a good number to ensure that a large fraction of the vertices are matched in each coarsening round. Instead of following exactly the same procedure as ParMetis, we add an important extension to the coarsening algorithm to specially handle 1-degree vertices. It has been observed that most real graphs follow power law distribution, which means a large number of vertices have very low degree. 1-degree vertices can be specially handled to improve the coarsening rate, by simply merging them into their only neighbor. Once again, this merge is done in two supersteps to resolve conflicts that arise if two vertices only connect to each other and nothing else.

Vertex Data Structure. The value associated with each vertex consists of the following four attributes: (1) *state* keeps track of which state the vertex is currently in. It can take one of the 4 values: NORMAL, MATCHREQUESTED, MERGED and MERGEHOST. NORMAL obviously indicates that the vertex is normal – ready to do any action; MATCHREQUESTED means that the vertex just sent out an match request; MERGED denotes that the vertex is or will be merged into another vertex; and MERGEHOST means that another vertex will be merged into this vertex. (2) *mergedTo* records the id of the vertex that this vertex is merged into, so that we can reconstruct the member vertices of a super vertex. This attribute is legitimate only when state=MERGED. (3) *weight* keeps track of the weight of the (super) vertex during the coarsening process. (4) *replacements* stores all the pair of vertex replacements in order to guarantee the correctness of graph structure change during a merge. For example, consider a graph where A is connected to B, which in turn links

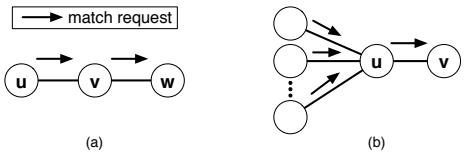


Figure 6: Examples in the matching phase of graph coarsening

to C. If B is matched and merged to A, B needs to notify C that C's connection to B will be replaced by a connection to A after the merge. In this case, $B \rightarrow A$ is stored in C's *replacements* before the actual merge happens. Besides vertex values, the value associated with an edge is the *weight* of the (super) edge.

After dealing with 1-degree vertices in the first 2 supersteps, the algorithm executes in iterations of m matching phases and a collapsing phase.

Matching Phase. Each matching phase consists of 2 supersteps: a match request step and a match notification step. In a match request step, the algorithm randomly scans over the vertices with *state*=NORMAL. For each such vertex u, it looks through the edges that are not in the *replacements* list (as vertices in *replacements* have already been matched), and chooses the one with the highest weight, say (u, v), by breaking ties arbitrarily. If v is local and $v.state$=NORMAL, we establish the match by setting $u.state$=MERGED, $u.mergedTo = v$ and $v.state$=MERGEHOST, and notifies each u's neighbor to add $u \rightarrow v$ in its *replacements* through either a local operation or a message depending on whether the neighbor is local or not. Otherwise, if v is remote, a match request message will be sent to v based on the ordering constraints. If a match request is sent, we set $v.state$=MATCHREQUESTED.

In a match notification step, when a vertex v receives match requests, we first check the *state* of v. If $v.state$ is MERGED or MERGEHOST, we ignore all the match requests, since v is already matched. If $v.state$ is MATCHREQUESTED, we also ignore the match requests. This is to avoid the conflict when v's match request to w is granted but at the same time v has already granted the match request from u. Under the definition of graph matching, no two matches should share a common vertex. This scenario is demonstrated in Figure 6(a). The ordering constraint doesn't help in this chain case, as the vertices could happen to be ordered $u < v < w$ and the phase number i could happen to be even. Therefore, we use $state$=MATCHREQUESTED to break chains. In summary, match requests to v are only considered when $v.state$=NORMAL. Then we choose the request with the heaviest edge weight, say from u, and send u a match notification. At the same time, we change $v.state$=MERGED and $v.mergedTo = u$, then notifies v's neighbors to add $v \rightarrow u$ in their *replacements*.

When a match notification is received by a vertex u at the beginning of the next match request step, we simply change $u.state$= MERGEHOST. On the other hand, if no match notification is received and $u.state$= MATCHREQUESTED, we change u's *state* back to NORMAL.

We have discussed above that a chain of match requests needs to be broken in order to avoid conflicts. As shown in Figure 6(b), for a hub vertex, potentially many other vertices could send match requests to it. However, as long as the hub vertex has issued a match request, all the requests that it receives will be ignored. In order to give a fair chance to all the match requests, we add a probability to decide whether a vertex u with $state$=NORMAL should send a match request or not. This probability is defined as $p_u = \frac{1}{log(1+|E_u|)}$, where $|E_u|$ is the number of edges incident on u. Based on this probability, a hub vertex is less likely to send a match request, and thus more likely to be the receiving end of match requests.

Collapsing Phase. After all the matches are established, in the collapsing phase, each vertex first processes all the *replacements* on the edges. After that, if an active vertex u has $state$=MERGED, it needs to be merged to the target vertex with id=$u.mergedTo$. If the target vertex is local, the merge is processed immediately. Otherwise, a merge request is sent to the target vertex with u's weight and all its edges with their weights. Next, we remove all u's edges but keep $u.state$ and $u.mergedTo$ so that later we can trace back who is merged into whom. After all the merges are done, if a vertex has $state$=MERGEHOST, we set it back to NORMAL to participate in the next round of coarsening.

5. A HYBRID MODEL

In the previous section, we have shown how the graph-centric model can benefit a number of graph algorithms. One of the major reasons for the improved performance under the graph-centric model is the allowance of the local asynchrony in the computation (e.g. the PageRank algorithm in Algorithm 4): a message sent to a vertex in the same partition can be processed by the receiver in the same superstep. In addition to using the flexible graph-centric programming model, asynchrony can also be achieved by a system optimization while keeping the same vertex-centric programming model. We call this approach as the *hybrid model*.

To implement the hybrid model, we differentiate the messages sent from one partition to the vertices of the same partition, called *internal messages*, from the ones sent to the vertices of a different partition, called the *external messages*. We keep two separate incoming message buffers for each internal vertex, one for internal messages called $inbox_{in}$, and one for the external messages called $inbox_{ex}$. The external messages are handled using exactly the same message passing mechanism as in the vertex-centric model. An external message sent in superstep i can only be seen by the receiver in superstep $i+1$. In contrast, an internal message is directly placed into $inbox_{in}$ and can be utilized immediately in the vertex's computation during the same superstep, since both the sender and the receiver are in the same partition. Suppose vertices A and B are in the same partition, and during a superstep i, A sends B an internal message M. This message is immediately put into B's $inbox_{in}$ in the same superstep (with proper locking mechanism to ensure consistency). If later B is processed in the same superstep, all messages in B's $inbox_{in}$ and $inbox_{ex}$, including M, will be utilized to perform B's *compute*() function. On the other hand, if B is already processed before M is sent in superstep i, then M will be kept in the message buffer until B is processed in the next superstep $i+1$. To reduce the overhead of maintaining the message buffer, we apply the *combine*() function on the internal messages, whenever a user-defined combiner is provided.

Under this hybrid model, we can keep exactly the same connected component algorithm in Algorithm 1 and the PageRank algorithm in Algorithm 3 designed for the vertex-centric model, while still benefiting from the asynchronous computation. However, one needs to be cautious when using the hybrid model. First of all, not all graph problems can benefit from asynchrony. Furthermore, blindly running a vertex-centric algorithm in the hybrid mode is not always safe. For example, the vertex-centric graph coarsening algorithm won't work under the hybrid model without change. This is because the graph coarsening algorithm requires different types of messages to be processed at different stages of the computation. The hybrid model will mix messages from different stages and confuse the computation. Even for PageRank, although our specially designed accumulative iterative update algorithm works without change in the hybrid model, the classic PageRank algorithm won't fly in the hybrid model without change. We point out that similar care also needs to be taken when designing algorithms in other systems that support asynchronous computation.

Note that GraphLab and Grace also allow asynchronous computation while keeping the vertex-centric model. Both systems achieve this goal mainly through the customization of different vertex scheduling polices in the systems. However, one price paid for supporting asynchrony in their systems is that the scheduler can not handle mutation of graphs. In fact, both GraphLab and Grace require the graph structure to be immutable. This, in turn, limits their applicability to any algorithm that mutates the structure of graphs, such as all the graph aggregation algorithms discussed in Section 4.3. In comparison, Giraph++ does not have such conflicts of interests: programmers can freely express the asynchronous computation in the graph-centric model, while the system maintains the ability to handle graph mutation.

6. EXPERIMENTAL EVALUATION

In this section, we empirically compare the performance of the vertex-centric, the hybrid, and the graph-centric models. For fair comparison among the different models, we implemented all of them in the same Giraph++ system. We refer to these implementations as Giraph++ Vertex Mode (VM), Hybrid Mode (HM), and Graph Mode (GM), respectively. We expect that relative performance trends that we observed in this evaluation will hold for other graph processing systems, though such study is beyond the scope of this paper.

6.1 Experimental Setup

We used four real web graph datasets, shown in Table 1, for all of our experiments. The first three datasets, uk-2002, uk-2005 and webbase-2001 were downloaded from `law.di.unimi.it/datasets.php`. These datasets were provided by the WebGraph [4] and the LLP [3] projects. The last dataset clueweb50m was downloaded from `boston.lti.cs.cmu.edu/clueweb09/wiki/tiki-index.php?page=Web+Graph`. It is the TREC 2009 Category B dataset. All four datasets are directed graphs. Since some algorithms we studied are for undirected graphs, we also converted the four directed graphs into undirected graphs. The numbers of edges in the undirected version of the graphs are shown in the 4th column of Table 1. These four dataset are good representative for real life graphs with heavy-tail degree distribution. For example, although the uk-2005 graph has an average degree of 23.7, the largest in-degree of a vertex is 1,776,852 and the largest out-degree of a vertex is 5,213.

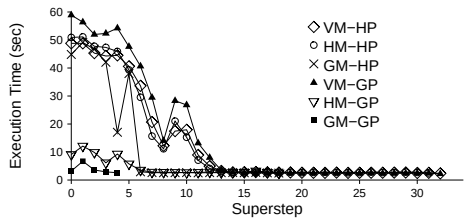
All experiments were conducted on a cluster of 10 IBM System x iDataPlex dx340 servers. Each consisted of two quad-core Intel Xeon E5540 64-bit 2.8GHz processors, 32GB RAM, and interconnected using 1Gbit Ethernet. Each server ran Ubuntu Linux (kernel version 2.6.32-24) and Java 1.6. Giraph++ was implemented based on a version of Apache Giraph downloaded in June 2012, which supports two protocols for message passing: Hadoop RPC and Netty (`https://netty.io`). We chose Netty (by setting -Dgiraph.useNetty=true), since it proved to be more stable. Each server was configured to run up to 6 workers concurrently. Since, a Giraph++ job requires one worker to be the master, there were at most 59 slave workers running concurrently in the cluster.

Note that in any of the modes, the same algorithm running on the same input will have some common overhead cost, such as setting up a job, reading the input, shutting down the job, and writing the final output. This overhead stays largely constant in VM, HM, and GM, regardless of the data partitioning strategy. For our largest dataset (webbase-2001), the common overhead is around 113 seconds, 108 seconds, and 272 seconds, for connected component, PageRank and graph coarsening, respectively. As many graph algorithms (e.g.

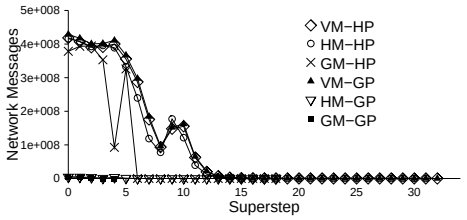
dataset	#nodes	directed #edges	undirected #edges	#partns	partitioning time (sec.)	directed				undirected			
						hash partitioned		graph partitioned		hash partitioned		graph partitioned	
						ncut	imblc	ncut	imblc	ncut	imblc	ncut	imblc
uk-2002	18,520,486	298,113,762	261,787,258	177	1,082	0.97	1.01	0.02	2.15	0.99	1.06	0.02	2.24
uk-2005	39,459,925	936,364,282	783,027,125	295	6,891	0.98	1.01	0.06	7.39	0.997	1.61	0.06	7.11
webbase-2001	118,142,155	1,019,903,190	854,809,761	826	4,238	0.97	1.03	0.03	3.78	0.999	1.41	0.03	5.05
clueweb50m	428,136,613	454,075,604	446,769,872	2891	4,614	0.9997	1.07	0.07	5.56	0.9997	1.96	0.06	6.95

Table 1: Datasets characteristics

PageRank and graph coarsening) requires 100s of supersteps, this cost is amortized quickly. For example, if we run 100 supersteps for PageRank and graph coarsening, then even for the fastest execution times on this dataset (GM with graph partitioning strategy), the common overhead accounts for around 7.7% and 8.7% of the total execution times, respectively. For the connected component algorithm, this cost is more noticeable, however the high-level trends are the same. Even with overheads included in the execution time, GM is 3X faster than VM on hash partitioned data (instead of 3.1X, if discounting the common overhead) for the connected component algorithm. Same speedup on graph partitioned data is 22X (instead of 63X, if discounting the common overhead). In order to focus on the difference in the processing part of VM, HM, and GM, we excluded the common overhead from the reported execution time for all the experiments. In addition, except for the experiments in Section 6.6, we turned off the checkpointing to eliminate its overhead.



(a) Execution Time



(b) Network Messages

Figure 7: The execution time and network messages per superstep for connected component detection on uk-2002 dataset.

6.1.1 Scalable Graph Partitioning

Graph partitioning plays a crucial role in distributed graph processing. For each experiment, we need to first decide on the number of partitions. Intuitively, a larger partition size would benefit GM and HM, as it increases the chance that neighboring vertices belong to the same partition. However, smaller partitions increase potential degree of parallelism and help balance workload across the cluster. We observed empirically that when each partition contained 100,000 to 150,000 vertices, all three modes performed well. Therefore, we heuristically set the number of partitions for each graph dataset so that it is a multiple of 59 (the number of slave workers) and each partition contained around 100,000 to 150,000 vertices. The number of partitions for each dataset used in our experiment is shown in column 5 of Table 1.

An even more important question is *how* to partition a graph. A good partitioning strategy should minimize the number of edges that connect different partitions, to potentially reduce the number of messages during a distributed computation. Most distributed graph processing systems, such as Giraph [1] and GraphLab [14], by default use random hash to partition graphs. Obviously, this random partitioning results in a large number of edges crossing partition boundaries. To quantify this property, we use the well-known Average Normalized Cut measure. Normalized Cut of a partition P, denoted *ncut(P)*, is defined as the faction of edges linking vertices in P to vertices in other partitions among all the outgoing edges of vertices in P, i.e. $\frac{|\{(u,v)\in E|u\in P\wedge v\notin P\}|}{|\{(u,v)\in E|u\in P\}|}$. The average ncut of all the partitions can be used to measure the quality of a graph partitioning. As shown in column 7 and column 11 of Table 1, the average ncuts for hash partitioning across different datasets are all very close 1, which means that almost all the edges cross partition boundaries.

GraphLab [14] and the work in [10], also proposed to employ the Metis [12] (sequential) or the ParMetis [11] (parallel) graph partitioning tools to generate better partitions. However, Metis and ParMetis cannot help when the input graph becomes too big to fit in the memory of a single machine.

We implemented a scalable partitioning approach based on the distributed graph coarsening algorithm described in Section 4.3.2. This algorithm mimics the parallel multi-level k-way graph partitioning algorithm in [11], but is simpler and more scalable. There are 3 phases in this algorithm: a coarsening phase, a partitioning phase, and a uncoarsening phase. In the coarsening phase, we apply the distributed graph coarsening algorithm in Section 4.3.2 to reduce the input graph into a manageable size that can fit in a single machine. Then, in the partitioning phase, a single node sequential or parallel graph partitioning algorithm is applied to the coarsened graph. We simply use the ParMetis algorithm in this step. At last, in the uncoarsening phase, we project the partitions back to the original graph. This phase does not apply any refinements on the partitions as in [11]. However, uncoarsening has to be executed in a distributed fashion as well. Recall that in the coarsening phase each vertex that is merged into another has an attribute called *mergedTo* to keep track of the host of the merger. This attribute can help us derive the membership information for each partition. Suppose that a vertex A is merged into B which in turn is merged into C, and finally C belongs to a partition P. We can use the *mergedTo* attribute to form an edge in a *membership forest*. In this example, we have an edge between A and B, and an edge between B and C. Finding which vertices ultimately belong to the partition P is essentially finding the connected components in the *membership forest*. Thus, we use Algorithm 2 in the last stage of graph partitioning.

Column 6 of Table 1 shows the total graph partitioning time, with graph coarsening running 100 supersteps in the graph-centric model. The average ncuts of graph partitions produced by this algorithm are listed in columns 9 and 13. On average, only 2% to 7% edges go across partitions.

There is another important property of graph partitioning that affects the performance of a distributed graph algorithm: load balance.

dataset	execution time (sec)						network messages (millions)						number of supersteps					
	hash partitioned (HP)			graph partitioned (GP)			hash partitioned (HP)			graph partitioned (GP)			hash partitioned (HP)			graph partitioned (GP)		
	VM	HM	GM	VM	HM	GM	VM	HM	GM	VM	HM	GM	VM	HM	GM	VM	HM	GM
uk-2002	441	438	272	532	89	19	3,315	3,129	1,937	3,414	17	8.7	33	32	19	33	19	5
uk-2005	1,366	1,354	723	1,700	230	90	11,361	10,185	5,188	10,725	370	225	22	22	16	22	12	5
webbase-2001	4,491	4,405	1,427	3,599	1,565	57	13,348	11,198	6,581	11,819	136	58	605	604	39	605	319	5
clueweb50m	1,875	2,103	1,163	1,072	250	103	6,391	5,308	2,703	6,331	129	69	38	37	14	38	18	5

Table 2: Total execution time, network messages, and number of supersteps for connected component detection

For many algorithms, running time for each partition is significantly affected by its number of edges. Therefore, we also need to measure how the edges are distributed across partitions. We define the *load imbalance* of a graph partitioning as the maximum number of edges in a partition divided by the average number of edges per partition, i.e. $\frac{\max(|E_P|)\times p}{\sum |E_P|}$, where $E_P = \{(u,v) \in E | u \in P\}$ and p is the number of partitions. Table 1 also shows the load imbalance factors for both hash partitioning and our proposed graph partitioning across different datasets. Clearly, hash partitioning results in better balanced partitions than our graph partitioning method. This is because most real graph datasets present a preferential attachment phenomenon: new edges tend to be established between already well-connected vertices [17]. As a result, a partition that contains a well-connected vertex will naturally bring in much more edges than expected. Producing balanced graph partitions with minimal communication cost is an NP-hard problem and is a difficult trade-off in practice, especially for very skewed graphs. Newly proposed partitioning technique with dynamic load balancing [24], and the vertex-cut approach introduced in the latest version of Graphlab [9] can potentially help alleviate the problem, but cannot completely solve the problem.

We decided to "eat our own dog food" and evaluate the connected component and PageRank algorithms, using our graph partitioning (GP) strategy in addition to the default hash partitioning (HP). Note that the focus of this evaluation is on comparing Giraph++ VM, HM, and GM modes. Although we implemented a scalable graph partitioning algorithm and use its output to evaluate two distributed graph algorithms, we leave the in-depth study of graph partitioning algorithms for the future work. The GP experiments should be viewed as a proxy for a scenario where some scalable graph partitioning algorithm is used as a part of graph processing pipeline. This should be the case if graphs are analyzed by (multiple) expensive algorithms, so that the performance benefits of the low ncut justify the partitioning cost.

6.2 Connected Component

The first algorithm we use for the performance evaluation is the connected component analysis. In VM and HM we implemented Algorithm 1, and in GM we implemented Algorithm 2.

Figure 7 shows the execution time and network messages per superstep as the connected component algorithm progresses on the uk-2002 dataset (the undirected graph version) under the three modes and with both hash partitioning (HP) and graph partitioning (GP) strategies. From this figure, we observe that: (1) The shapes of curves of the execution time look very similar to those of the network messages. Since the computation is usually very simple for graph traversal algorithms, execution time is mostly dominated by message passing overhead. (2) Better partitioning of the input graph doesn't help much in the VM case, because the vertex-centric model doesn't take advantage of the local graph structure. VM sends virtually the same number of network messages with GP and HP. The small differences are due to the use of the combiner. The total number of messages without combiner is always exactly the same in VM, no matter which partitioning strategy is used. (3) Under either HP or GP, GM performs significantly better as it generates much fewer network messages per superstep. Furthermore, GM reduces the number of supersteps needed for the algorithm. (4) GM benefits significantly from better partitioning of graphs, which allows it to do majority of the work in the first 3 supersteps.

HM only speeds up processing of messages local to a partition. Thus, it provides only marginal benefit under HP where 99% percent of edges go across partitions. However, under GP, where only 2% of edges cross partition boundaries, it provides 201X improvement in the number of messages, which translates into 6X improvement in overall running time. Still, GM does even better, as it uses algorithm-specific, per-partition data structures to store equivalent connected components. This allows GM to instantly change states of entire groups of vertices, instead of processing them one at a time. Thus, GM sends 2X fewer messages and runs 4.7X faster than HM.

The above observations also apply to the other 3 datasets as shown in Table 2. Across all the four datasets, GM dramatically reduces the number of iterations, the execution time and network messages. Under HP, the reduction in network messages ranges from 1.7X to 2.4X resulting in 1.6X to 3.1X faster execution time. The advantage of the graph-centric model is more prominent under GP: 48X to 393X reduction of network messages and 10X to 63X speedup in execution time.

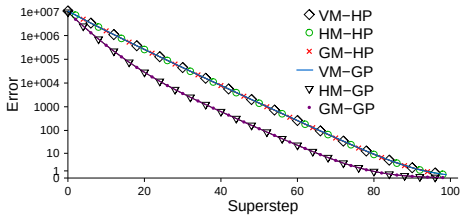
Note that a smaller number of network messages doesn't always translate to faster execution time. For example, for the uk-2005 dataset VM sends fewer messages under GP than under HP, yet it still runs 24% slower, due to load imbalance. The largest partition becomes the straggler in every superstep, thus increasing the total wall clock time.

6.3 PageRank

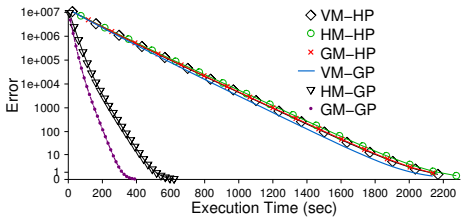
Next, we compare the performance of PageRank algorithm in different modes. In VM and HM we implemented Algorithm 3, and in GM we implemented Algorithm 4.

Figure 8 compares the convergence rates of PageRank on the uk-2002 dataset under different modes and partitioning strategies. The error is measured in L_1 error against the true PageRank values, obtained by running Algorithm 4 for 200 iterations. Under HP strategy, VM, HM and GM all exhibit very similar convergence behavior with supersteps and time (the corresponding three lines overlap in Figure 8(a) and are very close to each other in Figure 8(b)). This is because vertices in each partition mostly connect to vertices outside, thus there is very few asynchronous updates that actually happen in each superstep. As VM cannot take much advantage of better partitioning strategies, its convergence behavior under GP is still similar to that of HP. GM and HM, on the other hand, take significantly fewer supersteps and less time to converge under GP. Figure 8(b) shows dramatic reduction (more than 5X) of GM in time needed to converge, as compared to VM. HM converges much faster than VM, however its total time to converge is still up to 1.5X slower than GM, due to HM's generic, and hence less efficient, in-memory structures.

As the algorithms progress, the execution time and network messages per superstep on the uk-2002 dataset are depicted in Figure 9. The progression of PageRank looks very different than that of con-

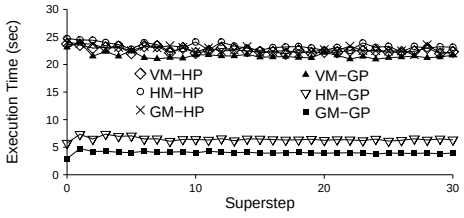


(a) Convergence with # supersteps

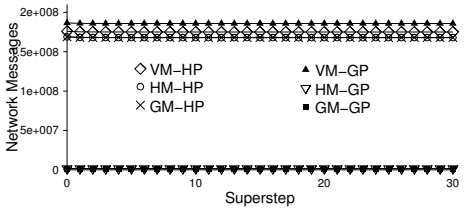


(b) Convergence with time

Figure 8: Convergence of PageRank on uk-2002 dataset



(a) Execution Time



(b) Network Messages

Figure 9: The execution time and network messages per superstep for PageRank on uk-2002 dataset (the first 30 supersteps).

nected component detection. No matter under which mode and which partitioning strategy, the execution time and network messages do not change much across different supersteps.

The performance comparisons are consistent across all four datasets, as shown in Table 3. When input graphs are better partitioned, GM brings in 41X to 116X reduction in network messages and 1.6X to 12.8X speedup in execution time per iteration, keeping in mind that it also results in much faster convergence rate. Again, due to the higher load imbalance resulted from the GP strategy, VM drags its feet for the uk-2005 dataset.

6.4 Graph Coarsening

We now empirically study the graph coarsening algorithm in VM and GM. Recall, that HM cannot be used for the algorithm implementation described in Section 4.3.2, as the algorithm does not allow asynchronous execution.

The implementation of the graph coarsening algorithm in VM follows a similar procedure as in GM, except that no local match

dataset	per-superstep execution time (sec)					
	hash partition (HP)			graph partition (GP)		
	VM	HM	GM	VM	HM	GM
uk-2002	22	23	23	21	6	4
uk-2005	88	91	89	167	28	13
webbase-2001	121	125	125	102	24	13
clueweb50m	62	64	65	34	21	21
dataset	per-superstep network messages (millions)					
uk-2002	175	168	168	186	1.6	1.6
uk-2005	701	684	684	739	6.7	6.7
webbase-2001	717	689	689	751	9.2	9.2
clueweb50m	181	181	181	168	4.1	4.1

Table 3: Execution time and network messages for PageRank

or merge operations could be performed and everything is done through message passing. Table 4 compares the two models in terms of execution time and network messages for running 100 supersteps of the graph coarsening algorithm. Note that we do not assume better partitioning of input graphs is available, since graph coarsening is usually the first step for a graph partitioning algorithm. As a result, all the input graphs are hash partitioned. Across all four datasets, the graph coarsening algorithm benefits at different levels from our graph-centric model, because some match and merge operations can be resolved locally without message passing. For the clueweb50m dataset, the execution time is more than halved. Figure 10 charts the progressive execution time and network messages per superstep for the clueweb50m dataset. Both time and messages fluctuate as the algorithm progresses. But since the coarsened graph continues to shrink, both measures decrease as a general trend. The coarsening rates on the clueweb50m dataset with both supersteps and time are shown in Figure 11. In terms of supersteps, the coarsening rates look neck and neck for VM and GM. Since clueweb50m is a sparser graph, the first 2 supersteps of handling 1-degree vertices are particularly effective in reducing the number of active vertices. Due to the better efficiency of GM, its coarsening rate with time excels (more than doubled).

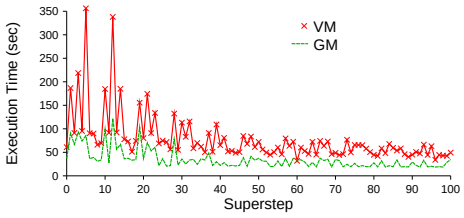
dataset	execution time (sec)		network messages (millions)	
	VM	GM	VM	GM
uk-2002	626	600	1,215	1,094
uk-2005	6,687	5,414	3,206	2,796
webbase-2001	3,983	2,839	4,015	3,904
clueweb50m	7,875	3,545	2,039	2,020

Table 4: Execution time and network messages for graph coarsening running 100 supersteps

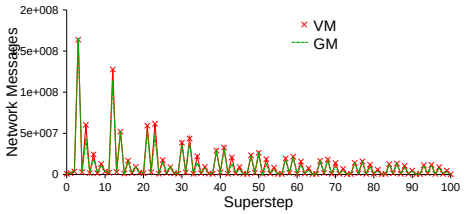
6.5 Effect of Partitioning Quality

We now investigate the effect of partitioning quality on the performance of different models. We chose the uk-2002 dataset for this experiment. Besides HP and GP partitioning strategies, we constructed a different partitioning (abbreviated as SP) of the uk-2002 graph by randomly swapping 20% vertices in different partitions of the GP strategy. The resulting partitions have average ncut 0.57 and 0.56 for the undirected version and the directed version of the graph, respectively. The corresponding imbalance factors are 1.83 and 1.75. Both the average ncuts and the imbalance factors of SP fall between those of HP and GP. Table 5 lists the execution times of connected component and PageRank with the three partitioning strategies. The results are consistent with previous experiments. Better partitioning of input doesn't help much for VM. In fact, VM under GP performs worse than under SP due to higher imbalance. In contrast, both HM and GM benefit a lot from better-quality partitioning strategies, with GM benefiting the most.

6.6 Effect on Fault Tolerance

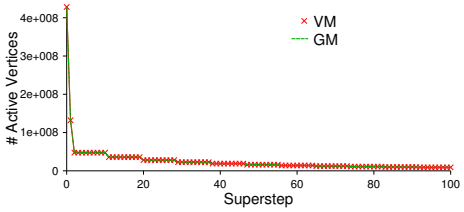


(a) Execution Time

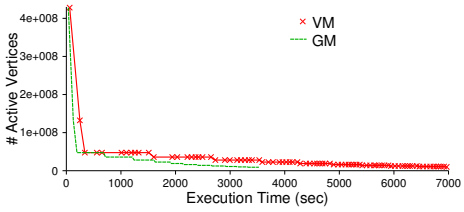


(b) Network Messages

Figure 10: Execution time and network messages per superstep for graph coarsening on clueweb50m dataset.



(a) Coarsening with # Supersteps



(b) Coarsening with time

Figure 11: Coarsening rates on clueweb50m dataset.

In Section 3, we have discussed that Giraph++ utilizes the same checkpointing and failure recovery mechanism as in Giraph to ensure fault tolerance. The only difference is that in GM mode Giraph++ also stores and reads back the auxiliary data structures for each graph partition (e.g. equiCC for the connected component algorithm), respectively. Sometimes, GM mode also introduces more attributes for each vertex, such as the delta value in the PageRank algorithm. It may seem that a lot of overhead is introduced in GM. However, this little bit of extra cost is sometimes outweighed by the dramatic reduction of messages and number of iterations. During our empirical study, we consistently observed similar or even reduced overhead for checkpointing and failure recovery under GM. Let's take the relatively large webbase-2001 dataset as an example. If we turn on checkpointing for every 10th superstep for the PageRank computation, with hash-partitioned input, 59 workers spend collectively 7,413 seconds for checkpointing under VM and 7,728 seconds under GM (the elapsed time is roughly 7728/59=131 seconds) for running 30 supersteps. In comparison, with better

	connected component total time (sec)			pagerank per-superstep time (sec)		
	HP	SP	GP	HP	SP	GP
VM	441	356	532	22	18	21
HM	438	199	89	23	13	6
GM	272	106	19	23	12	4

Table 5: Effect of partitioning quality on uk-2002 dataset

	Giraph++ (sec)			GraphLab (sec)	
	VM	HM	GM	Sync	Async
HP	4,491	4,405	1,427	912	error
GP	3,599	1,565	57	150	161

Table 6: Connected component in Giraph++ and GraphLab

graph partitions, VM takes 7,662 seconds, whereas GM spends 6,390 seconds collectively for the same other setting. For the graph coarsening algorithm, VM takes totally 28,287 seconds for checkpointing with 59 workers running 100 supersteps (checkpointing for every 10th superstep), whereas GM takes about 27,309 seconds. Furthermore, as GM often results in fewer iterations, the number of checkpoints needed for an algorithm is also reduced. For connected component detection, which only takes 10s of supersteps under GM, it is not necessary to even turn on the checkpointing option. Similar behavior was observed for reading the checkpointed state during recovery.

6.7 Comparison with Prior Work

Finally, we compare Giraph++ with GraphLab for the connected component algorithm on the largest webbase-2001 dataset. Once again, we want to emphasize that the focus of this paper is on the flexible graph-centric programming model and that it can be implemented in different graph processing systems. Head-to-head comparison between different systems is not the purpose of this paper, as the different implementation details (e.g. Giraph++ uses Netty or Hadoop RPC for communication, whereas GraphLab uses MPI) and the use of different programming languages (Giraph++ in Java and GraphLab in C++) contribute a great deal to the difference in performance. Nevertheless, the performance numbers in Table 6 shed some light on the potential of the graph-centric programming model in other graph processing systems.

In this experiment, we ran GraphLab (version 2.1) using the same number of workers on the same 10-node cluster as described in Section 6.1. We also use the same hash partitioning (HP) and graph partitioning (GP) strategies described in Table 1 for Giraph++ and GraphLab. The reported execution time for both systems excludes the common overhead of setting up jobs, reading inputs, writing outputs and shutting down jobs. As Table 6 shows, the synchronous mode (denoted as Sync) in GraphLab has much more efficient runtime compared to the equivalent VM mode in the Giraph++ BSP model, partly due to the system implementation differences outlined above. GraphLab also benefits from the better graph partitioning strategy. However, the asynchronous mode (denoted as Async in Table 6) in GraphLab did not deliver the better performance as expected. Under HP, Async ran into memory allocation error, and under GP it performed worse than Sync. Even though Async often needs fewer operations to converge, it also reduces the degree of parallelism due to the locking mechanism needed to ensure data consistency. For the connected component algorithm on this particular dataset, this trade-off results in the inferior performance of Async. Nevertheless, the most important takeaway from this experiment is that GM in Giraph++ even outperforms the best of GraphLab by 2.6X, despite the system implementation differences. This result further highlights the advantage of the graph-centric programming model and also shows its potential in other graph processing systems.

7. RELATED WORK

An overview of Pregel[15] and Giraph[1] is provided in Section 2.

GraphLab [14] also employs a vertex-centric model. However, different from the BSP model in Pregel, GraphLab allows asynchronous iterative computation. As another point of distinction, Pregel supports mutation of the graph structure during the computation, whereas GraphLab requires the graph structure to be static. This limits GraphLab's application to problems like graph coarsening [11], graph sparsification [19] and graph summarization [22].

Kineograph [6] is a distributed system for storing continuously changing graphs. However, graph mining algorithms are still performed on static snapshots of changing graphs. Kineograph's computation model is also vertex centric.

Trinity Graph Engine [20] handles both online and offline graph processing. For online processing, it keeps the graph topology in a distributed in-memory key/value store. For offline processing, it employs the similar vertex-based BSP model as in Pregel.

Grace [23] is a single-machine parallel graph processing platform. It employs the similar vertex-centric programming model as in Pregel, but allows customization of vertex scheduling and message selection to support asynchronous computation. However, Grace, too, requires immutable graph structure.

Naiad [16] is designed for processing continuously changing input data. It employs a differential computation model and exposes a declarative query language based on .NET Language Integration Query (LINQ).

8. CONCLUSION

In summary, we proposed a new graph-centric programming model for distributed graph processing and implemented it in a system called Giraph++. Compared to the vertex-centric model used in most existing systems, the graph-centric model allows users to take full advantage of the local graph structure in a partition, which enables more complex and flexible graph algorithms to be expressed in Giraph++. By exploiting the use of the graph-centric model in three categories of graph algorithms – graph traversal, random walk and graph aggregation – we demonstrated that the graph-centric model can make use of the off-the-shell sequential graph algorithms in distributed computation, allows asynchronous computation to accelerate convergence rates, and naturally support existing partition-aware parallel/distributed algorithms. Furthermore, Giraph++ is able to support both asynchronous computation and mutation of graph structures in the same system.

Throughout our empirical study, we observed significant performance improvement from the graph-centric model, especially when better graph partitioning strategies are used. The better performance of the graph-centric model is due to the significant reduction of network messages and execution time per iteration, as well as fewer iterations needed for convergence. This makes the graph-centric model a valuable complement to the existing vertex-centric model. In the future, we would like to explore more graph algorithms using this model and conduct more in-depth study of distributed graph partitioning algorithms.

9. REFERENCES

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

[2] A. Balmin, V. Hristidis, and Y. Papakonstantinou. Objectrank: authority-based keyword search in databases. In *VLDB'04*, pages 564–575, 2004.

[3] P. Boldi, M. Rosa, M. Santini, and S. Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *WWW'11*, pages 587–596, 2011.

[4] P. Boldi and S. Vigna. The WebGraph framework I: Compression techniques. In *WWW'04*, pages 595–601, 2004.

[5] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *WWW'98*, pages 107–117, 1998.

[6] 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'12*, pages 85–98, 2012.

[7] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *OSDI'04*, pages 137–150, 2004.

[8] A. Frommer and D. B. Szyld. On asynchronous iterations. *J. Comput. Appl. Math.*, 123:201–216, 2000.

[9] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: distributed graph-parallel computation on natural graphs. In *OSDI'12*, pages 17–30, 2012.

[10] J. Huang, D. J. Abadi, and K. Ren. Scalable sparql querying of large rdf graphs. *PVLDB*, 4(11):1123–1134, 2011.

[11] G. Karypis and V. Kumar. A coarse-grain parallel formulation of multilevel k-way graph partitioning algorithm. In *SIAM PP'97*, 1997.

[12] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, 1998.

[13] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. *J. ACM*, 46(5):604–632, 1999.

[14] 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. *PVLDB*, 5(8):716–727, 2012.

[15] 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'10*, pages 135–146, 2010.

[16] F. McSherry, D. G. Murray, R. Isaacs, and M. Isard. Differential dataflow. In *CIDR'13*, 2013.

[17] M. E. J. Newman. Clustering and preferential attachment in growing networks. *Phys. Rev. E*, 64:025102, 2001.

[18] P. S. Pacheco. *Parallel programming with MPI*. Morgan Kaufmann, 1997.

[19] V. Satuluri, S. Parthasarathy, and Y. Ruan. Local graph sparsification for scalable clustering. In *SIGMOD'11*, pages 721–732, 2011.

[20] B. Shao, H. Wang, and Y. Li. Trinity: A Distributed Graph Engine on a Memory Cloud. In *SIGMOD'13*, pages 505–516, 2013.

[21] I. Stanton and G. Kliot. Streaming graph partitioning for large distributed graphs. In *KDD'12*, pages 1222–1230, 2012.

[22] Y. Tian, R. Hankins, and J. M. Patel. Efficient aggregation for graph summarization. In *SIGMOD'08*, pages 419–432, 2008.

[23] G. Wang, W. Xie, A. Demers, and J. Gehrke. Asynchronous large-scale graph processing made easy. In *CIDR'13*, 2013.

[24] S. Yang, X. Yan, B. Zong, and A. Khan. Towards effective partition management for large graphs. In *SIGMOD'12*, pages 517–528, 2012.

[25] Y. Zhang, Q. Gao, L. Gao, and C. Wang. Accelerate large-scale iterative computation though asynchronous accumulative updates. In *ScienceCloud'12*, pages 13–22, 2012.