An Efficient Graph Processing System

Xianke Zhou, Pengfei Chang, and Gang Chen

College of Computer Science and Technology
Zhejiang University, Hangzhou, China
{xiankz,changpeng3336,cg}@zju.edu.cn

Abstract. Conventional graph processing algorithms are not designed for those unprecedented large graphs and result in suboptimal performance. To address the problem, Google proposed its Pregel system, which adopts a vertex-centric processing framework for simplifying the development of parallel graph algorithms. In Pregel, the graph computation proceeds iteratively and each iteration is called a superstep. Pregel's processing engine adopts the Bulk Synchronous Parallel (BSP) model, which simplifies the synchronization mechanism and ensures that the system reaches a global synchronization at the end of each superstep. This strategy however significantly increases the system overhead for algorithms that entail many iterations. In this paper, we propose a new graph processing framework based on Pregel. It extends Pregel by introducing a new data structure, *super-vertex*, and a new API, *internalCompute*. Our system is fully compatible with Pregel in that the codes of Pregel can run on it without modification. Moreover, we allow the programmers to optimize their codes with the unique two-phase processing strategy. We evaluated the advantages of our approach by two popular graph algorithms, Shortest Path and PageRank, with real dataset from twitter.

1 Introduction

MapReduce [2] has been widely adopted in various big data applications. Its simple but flexible interface allows the programmers to develop high-performance parallel algorithms without delving into the scheduling, synchronization and other non-trivial implementation issues. However, as argued in [8][9], MapReduce is inefficient in processing graph data. Consequently, Google introduces an alternative system called Pregel [9], which is specifically designed for large-scale graph processing.

Pregel is based on the vertex-centric computation model. The vertices communicate with each other via messages. It defines the *Compute* function to process the incoming messages, and adopts Bulk Synchronous Parallel (BSP) model [14] as a means to synchronize the processing where the computation is split into multiple iterations (aka superstep). In each superstep, the vertex retrieves its incoming messages and processes them sequentially. Pregel monitors the status of the vertices. If all vertices consume their messages of a given step and finish the processing, it can progress to a new superstep. In this way, a global synchronization is achieved at the end of each iteration.

In Pregel, except the CPU cost for processing the *Compute* function, the main cost consists of the message forwarding cost and synchronization cost. We use Figure 1 to demonstrate the cost estimation. Suppose we have two workers (the compute nodes),

L. Chen et al. (Eds.): APWeb 2014, LNCS 8709, pp. 401–412, 2014.

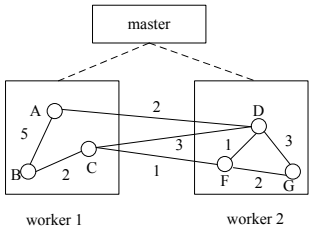

Fig. 1. Example of Graph Processing

and the graph vertices are randomly distributed to the workers. To compute the shortest path from *B* to the other vertices, *B* forwards the initial message to *A* and *C* in the first superstep. In the second superstep, *A* and *C* become active and update their paths to *B* accordingly. To broadcast the message, *C* forwards its shortest path to *D* and *F*, while *A* notifies *D* about its new path. Vertex *D* and *F* are hosted by the other worker. Therefore, the communication incurs network overhead. At the end of superstep 2, *A* and *C* turn inactive and vote to stop. The process has three drawbacks for the first two supersteps.

1. Vertex *C* sends the same message to *D* and *F*. As both *D* and *F* are located in worker 2, *C* generates redundant messages between worker 1 and worker 2.
2. Both vertex *A* and *C* forward their shortest paths to *D*. However, as *C*'s path is better than *A*'s, it is not necessary for *A* to notify *D*.
3. The first two supersteps only involve vertices in worker 1. In fact, we can combine them into one superstep to reduce the synchronization cost.

Moreover, at the end of each superstep, the workers communicate with the master to reach a synchronized status (whether to start the next superstep or stop). Such synchronization is costly, as straggling workers may cause the delay of the whole system in reaching a global synchronization point. Some complex algorithms may require a few hundreds of supersteps, and the synchronization cost is likely to dominate the total processing cost. If we can reduce the number of required supersteps, we can significantly boost the performance of such algorithms.

In this paper, we propose P++, an improved graph processing framework based on the Pregel. P++ defines a new data structure, *super-vertex*, and a new processing interface, *internalCompute*. The super-vertex represents a set of connected vertices. It maintains a subgraph for those vertices. A super-vertex can dynamically shrink or expand by removing or adding new vertices, and multiple super-vertices that are connected can be merged together. By introducing the concept of super-vertex, the computation is split into two phases in each superstep. In phase one, we apply the *internalCompute* function to process the data inside each super-vertex. In phase two, the original *Compute* function of Pregel is invoked to continue the computation for the whole super-vertex.

The remainder of the paper is organized as follows. In Section 2, we propose our new framework P++ and compare it with Pregel via Shortest Path graph algorithm. We evaluate the new framework in Section 3 and review the related work in Section 4. The paper is concluded in Section 5.

2 Computation Model of P++

Pregel adopts a vertex-centric computation model, where all interfaces are defined for the vertices. Table 1 lists the main interfaces provided by Pregel. The user-defined processing logic is typically implemented in the *Compute* function. For the space limitation, the detail description of Pregel is referred to [9][14][4].

Table 1. Pregel Interfaces

Function	Description
Compute(msgList)	process the messages
SendMessageTo(destV, &msg)	send message to neighbor vertex
VoteToHalt()	vote to be inactive
superstep()	get current superstep number
GetValue()	get the vertex value
GetOutEdgeIterator()	get the out edges

To reduce the message cost and synchronization cost, we design our new framework P++ by extending the open-source implementation of Pregel, GPS[10]. P++ is compatible with the Pregel's interface and the users' programs can therefore run in the new framework without modifications. In P++, we introduce a two-phase processing model and a new interface *internalCompute*. *internalCompute* is designed for the new compute unit, *super-vertex*, which represents a set of connected vertices.

2.1 Interface of P++

In parallel graph processing, the large graph is typically partitioned into subgraphs [11][17]. Each subgraph is assigned to a compute node and all compute nodes start their processing in parallel. Based on the same philosophy, we define a new concept, super-vertex, for the P++ processing framework.

Definition 1. **Connected Subgraph**
For the graph $G = (V,E)$, its subgraph $G' = (V',E')$ is a connected subgraph, if

1. *$V' \subset V$ and $E' \subset E$*
2. *$\forall v_i \in V', \forall v_j \in V'$, there is a path $v_i \rightsquigarrow v_0 \rightsquigarrow \ldots \rightsquigarrow v_n \rightsquigarrow v_j$ and $v_x \in V'$ for $0 \leq x \leq n$.*

Definition 2. **Super-Vertex**
For a connected subgraph $G' = (V',E')$, we define a super-vertex S, which represents all vertices in V'. S maintains two types of edges, internal edge E_{in} and external edge E_{ex}. $E_{in} = E'$, while E_{ex} is defined as:

- *If there is an edge $e = (v_i, v_j)$ and $v_i \in V' \wedge v_j \notin V'$, then we use a new edge $e' = (S,S')$ to replace e, where S' is the super-vertex for v_j. $e' \in E_{ex}$.*

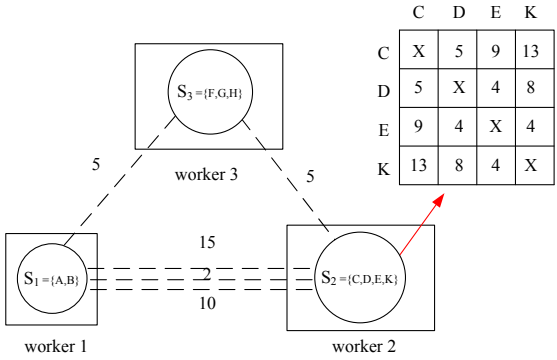

Fig. 2. Super-Vertices

For the example of Shortest Path algorithm, we transform the subgraph of each worker to a super-vertex and the result is shown in Figure 2. The external edges of super-vertices are the edges of the new graph. Note that there are three edges between super-vertices S_1 and S_2, as they are linked by three external edges representing $A \rightsquigarrow D$, $B \rightsquigarrow C$ and $B \rightsquigarrow D$, respectively. Inside each super-vertex, we keep the structure of the corresponding subgraph, namely the vertices and internal edges of the subgraph. The transformation allows us to use the super-vertex as the processing unit in Pregel. Note that in real scenario, each worker can have multiple connected subgraphs and hence multiple super-vertices are created.

The super-vertex can expand or shrink adaptively via the *merge* and *split* operations.

Definition 3. **Super-Vertex Merge**
Two super-vertices S_i and S_j can be merged, if there exist edges between them. Let G_i and G_j denote the subgraphs of S_i and S_j, respectively. The merged super-vertex S is generated for the new subgraph $G = (G_i.V \cup G_j.V, G_i.E \cup G_j.E \cup E_{ij})$, where E_{ij} represents the edges between G_i and G_j.

split is a reverse operation of *merge*, which partitions the subgraphs into two disjoint connected subgraphs. The sizes of the subgraphs are configurable. By applying the *merge* and *split* operations, we can adaptively tune the size of the super-vertex. To communicate between super-vertices, we define the super-message as:

Definition 4. **Super-Message**
Super-message follows the format of

$$(\{(m, \{v_0, ..., v_n\}), (m', \{v'_0, ..., v'_n\}), ...,\}, S)$$

where m is the message value, v_i denotes the ID of a vertex and S is the receiver's super-vertex ID.

In one super-message, the super-vertex can send different messages to multiple vertices from the same super-vertex. Moreover, the message value can be shared among the vertices. For example, the message from S_1 to S_2 in Figure 2 is represented as:

$$(\{(20, \{C\}), (12, \{D\}), (15, \{D\})\}, S_2)$$

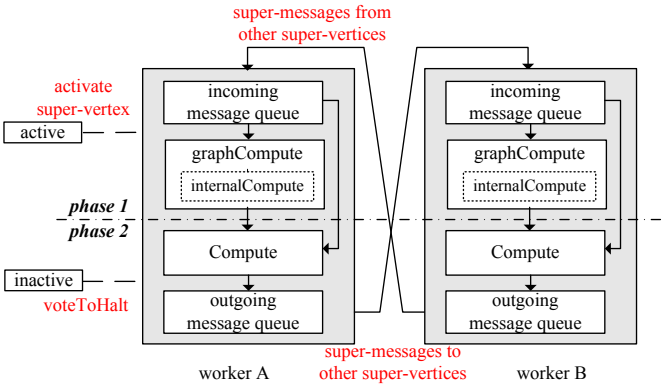

Fig. 3. Two-Phase Model

2.2 Two-Phase Processing Model

If we adopt the super-vertex as the processing unit, the computation within each superstep will be performed in two phases. As shown in Figure 3, the incoming super-messages from the other super-vertices are buffered in the queue and if the super-vertex is inactive, it will be woken up.

Algorithm 1. graphCompute(Iterable<M> supermsgs)

```
1: int microstep = 1
2: while !isStop() do
3:     for every vertex v in this.subGraph do
4:         if microstep==1 then
5:             internalCompute(v, getMessage(supermsgs,v), 1)
6:         else
7:             internalCompute(v,getMessageFromBuffer(v), microstep)
8:     microstep++
```

In phase 1, the messages are passed to the *graphCompute* function, which invokes the new *internalCompute* interface to process the subgraph of the corresponding super-vertex. The algorithms can update the internal vertices and their values, but cannot interact or access the data in other super-vertices. *internalCompute* follows the same definition as Pregel's *Compute* interface. In most cases, users can directly copy their codes from the *Compute* function to the *internalCompute* function. The only difference between the two interfaces is that they perform computation for the vertex and super-vertex, respectively. *graphCompute* specifies how the subgraph is processed. Although users can define their own *graphCompute* function, we recommend them to use the default implementation, which is shown as Algorithm 1.

graphCompute splits the processing into micro-steps. In the first micro-step, each vertex processes messages received from other super-vertices in the last super-step and

send messages to its neighbors. In the following micro-steps, the vertex continues the processing if it receives messages from vertices within the same super-vertex. Note that there is no synchronization requirement for the micro-step and the messages are passed to different vertices using the memory buffer. The whole process terminates when no message is generated any more.

In phase 2, the super-vertex continues the processing by the *Compute* interface. It exploits the partial results of the *graphCompute*. If one super-vertex needs to communicate with others, it generates the outgoing super-messages, which will be processed in the next superstep. At the end of the *Compute* function, the super-vertex can vote to stop, if all its tasks are done.

Figure 4 lists the interfaces of the super-vertex (based on GPS [10]). Some inherited interfaces from Pregel, such as *SendMessageTo*, are shared between *Vertex* and *SuperVertex* class and are discarded in the declarations.

```
public abstract class  SuperVertex<V extends MinaWritable,
E extends MinaWritable, M extends MinaWritable> extends Vertex{

   private Graph subGraph;

   public Graph getSubGraph(){return subGraph;}
   protected abstract void graphCompute(Iterable<M> messageValues);
   public abstract void merge(SuperVertex neighbor);
   public abstract SuperVertex split();
   //for subgraph processing
   public abstract void internalCompute(Vertex v, Iterable<M> messageValues,
                                        int microstepNo);
   @Override
   public abstract void compute(Iterable<M> messageValues,
                        int superstepNo);
}
```

Fig. 4. Interface of Super-Vertex

2.3 Shortest Path Processing

In the shortest path algorithm, we use Figure 2 to illustrate how P++ processes the graph data. The *internalCompute* function (Algorithm 2) defines how the normal vertex in a super-vertex performs its computation. In fact, we follow the same processing logic as the shortest path algorithm in Pregel. Specifically, the programmers can copy their codes for previous *Compute* function to the *internalCompute* function with a little change for using the new argument *v*. In the initial micro-step, the vertex broadcasts a new path to its neighbors (line 1-4). In the following micro-steps, the vertex updates its shortest path to the other vertices inside the same super-vertex progressively (line 6-13). The *SendMessageTo* (line 4 and 13) buffers the messages in memory, instead of forwarding them via the network connection.

Note that between two consecutive micro-steps, no synchronization is needed. *graphCompute* function (Algorithm 1) defines how the computation is scheduled. *graphCompute* checks the message buffer before starting a new micro-step. If there are messages inside the buffer, a new micro-step will start and *internalCompute* will be invoked immediately. Otherwise, *graphCompute* will terminate its processing. As a result, we can

obtain the shortest distance between every pair of graph vertices in a super-vertex. Figure 2 shows the result of super-vertex S_2, when *graphCompute* terminates.

Algorithm 2. internalCompute(Vertex v, Iterable<M> msgs, int mno)

```
1: if mno == 1 then
2:     for every neighbor vertex v_i of v do
3:         Path P = new Path(v, v_i, getWeight(v, v_i))
4:         SendMessageTo(v_i, P)
5: else
6:     for M msg : msgs do
7:         Path P = (Path)msg
8:         Path P' = getShortestPath(P.root)
9:         if P'.weight> P.weight then
10:            v̄.setShortestPath(P.root, P)
11:            for every neighbor vertex v_i of v do
12:                Path nextP = new Path(P'.root, v_i, getWeight(v, v_i))
13:                SendMessageTo(v_i, nextP)
```

In the processing, each super-vertex may receive multiple shortest paths from the last superstep. It will iteratively generate all new paths and only the shortest one is selected. More formally, if super-vertex S_i receives a path set P from the last superstep, it can compute its shortest paths as follows. Let $g(p)$ return the last vertex in a path p. We retrieve a corresponding subset of vertices $\theta(g(p))$ from S_i. Vertices in $\theta(g(p))$ are directly connected to $g(p)$ with an edge. We use $f(v_k, v_j)$ to denote the shortest path from v_k to v_j, which is computed in *graphCompute*. The shortest paths of S_i can be computed as:

$$\{min(p \circ f(v_k, v_j))|\forall p \in P, \forall v_k \in \theta(g(p)), \forall v_j \in S_i\}$$

where *min* returns the path with the shortest distance and $\circ$ denotes the concatenation of two paths.

By adopting the two-phase processing strategy, we reduce the number of supersteps to three and the inter-vertex messages to four, which is a significant improvement over the original algorithm. The key difference is that by iteratively evoking *internalCompute* function, we obtain the shortest distance paths for every pair of inside vertices, instead of waiting for the next superstep to generate the results. Compared to the original Pregel, the number of supersteps is reduced from $O(N)$ to $O(N')$ in P++, where N and N' are the number of graph vertices and super-vertices, respectively.

3 Performance Evaluation

To show the superior performance of P++, we implement the Shortest Path algorithm discussed in Section 3 and PageRank algorithm from [15], with the real dataset from Twitter. Twitter dataset has about 40 million vertices and more than 1 billion edges [5]. The user profiles were crawled from July 6th to July 31st, 2009. For the dataset,

Table 2. Experiment Settings

Parameter	Range and Default Value
Cluster Size	50
Memory Buffer per Super-Vertex	10M
Twitter Graph Size	4M-32M (32M)
HDFS Chunk Size	64M

we generate some synthetic datasets by only using the first K vertices and their edges. The synthetic dataset was used to evaluate the scalability of P++. The experiments were conducted on our in-house cluster. Each cluster node (machine) is powered with an intel Xeon 2.4GHZ CPU, 8GB memory and 512GB disk. All cluster nodes are connected via a high-speed 1GB Cisco switch. To reduce resource contention, each cluster node only hosts one P++ worker. The metrics used in the experiments are processing time and number of supersteps. Table 2 lists the configurations of the cluster and the experiments.

3.1 Shortest Path

In the shortest path algorithm, we randomly select a vertex as the root and compute the shortest pathes from the other vertices to the root. In above experiments, we show the average performance. In fact, the selection of root significantly affects the performance. Some root has many out-edges and thus triggers a large number of messages. Some root has few neighbors and the pathes are easy to compute.

Figure 5 and 6 show the effect of data size. P++ is more scalable and provide a stable performance. The processing time of both approaches does not increase linearly with the data size. Because increasing data size in Twitter does not necessarily lead to a higher processing cost for those users, as their reachable users are limited.

In Figure 7 and Figure 8, the performance of GPS is affected by the root selection. Twitter has a skewed user-base. Some popular users will incur very high computation cost, while some inactive users only require a few super-steps. By employing P++, we can effectively reduce the performance variance. The popular users and their friends are grouped into one super-vertex, where the computation is performed by *internalCompute*. The data skewness is neutralized by the adoption of super-vertex structure.

3.2 PageRank

P++ is fully compatible with the Pregel (GPS) interface. If the users does not define the *internalCompute*, P++ will adopt the original processing logic defined in the *Compute* function. But P++ still provides a better performance because P++ groups vertices into super-vertices and no message exchange is required inside a super-vertex. To illustrate the benefit of P++, we use *GPS-P++* to denote the performance of original GPS codes running on P++. We remove the diagram of GPS, as it always performs worse than GPS-P++. In the PageRank computation, we set a stop threshold ε. When the changes of rank values of all vertices are bounded by ε in a consecutive super-step, we assume that the algorithm converges and terminate the computation. On the other hand, we set a maximal super-step number S. If the PageRank algorithm does not converge after N super-steps, we will stop it forcedly. The initial rank value of a vertex is set to $\frac{1}{N}$, where

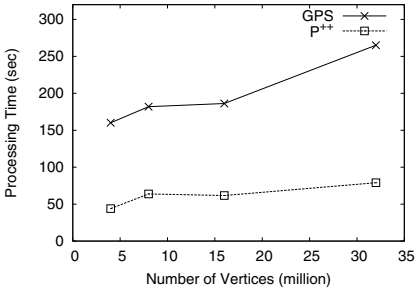

Fig. 5. Shortest Distance for Varied Data Size (Processing Time)

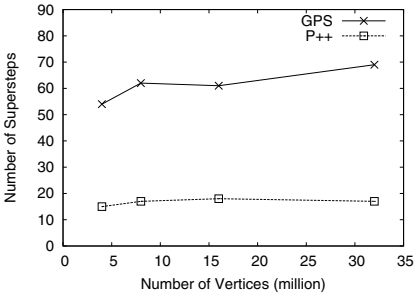

Fig. 6. Shortest Distance for Varied Data Size (# of Super-Steps)

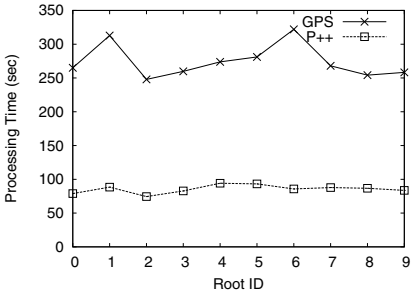

Fig. 7. Shortest Distance for Varied Roots (Processing Time)

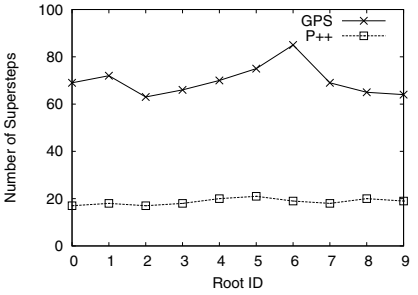

Fig. 8. Shortest Distance for Varied Roots (# of Super-Steps)

N is the total number of vertices in the dataset. In the experiments, the default values of ε and S are $10E-8$ and 50, respectively.

Figure 9 and Figure 10 show the effect of data size. It costs more time for P++ to process a large graph, because its *internalCompute* needs to handle more local PageRank computation. However, the performance of GPS-P++ seems to be less affected by the data size. In fact, that is because GPS-P++ cannot converge before 50 iterations. We stop it before it can provide the satisfied PageRank results.

In Figure 11 and Figure 12, we vary the threshold from $10E-3$ to $10E-9$. Performance of P++ is only slightly affected by the threshold, as the local PageRank values are computed in the *internalCompute*. GPS-P++ can terminate faster for a loose threshold, as it triggers few super-steps before completion. However, when the threshold becomes tighter, the performance of GPS-P++ drops dramatically. Clearly, for a tight stop threshold, P++ shows superior performance than GPS-P++.

3.3 Comparison with GraphLab

Due to the recent interest generated by asynchronized processing model, we also compare the performance of P++ with GraphLab. In this experiment, we only run the PageRank algorithm to measure the loading cost and query processing cost. Figure 13 shows the results. P++ incurs less loading cost, as its partitioning technique is well

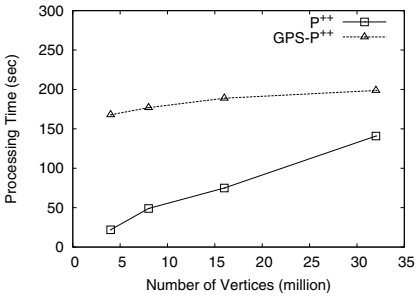

Fig. 9. PageRank for Varied Data Size (Processing Time)

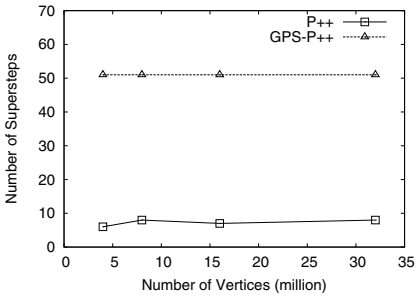

Fig. 10. PageRank for Varied Data Size (# of Super-Steps)

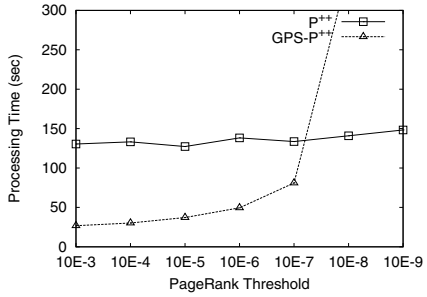

Fig. 11. PageRank for Varied Threshold (Processing Time)

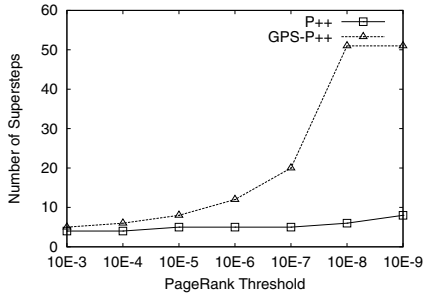

Fig. 12. PageRank for Varied Threshold (# of Super-Steps)

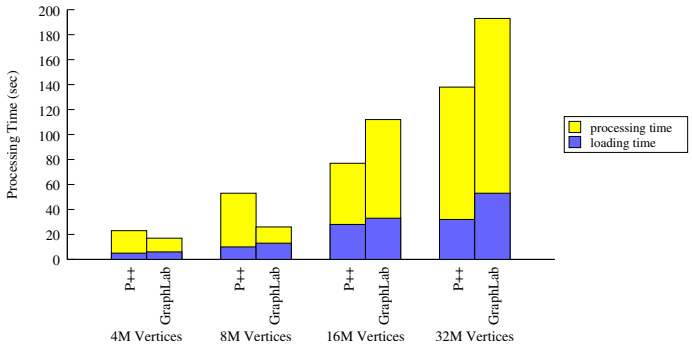

Fig. 13. P++ VS GraphLab of PageRank

integrated with Pregel, whereas GraphLab uses a sophisticated vertex-cut partitioning approach [3]. For the PageRank algorithm, P++ outperforms GraphLab for large datasets. Its two-phase strategy speeds up the convergence of the PageRank algorithm and no synchronization cost is incurred for computing the local PageRank values of

each subgraph. In our ongoing work, we are extending and incorporating the P++'s two-phase model into GraphLab.

4 Related Work

To process large-scale graphs, previous work transformed the graph algorithms into a set of MapReduce jobs[1] which are executed sequentially. However, MapReduce is not designed for such iterative processing. For this purpose, Google introduced its Pregel [9] for large-scale graphs processing. In Pregel, the programmers write codes for each vertex. Many graph algorithms, such as PageRank and KMeans, can be implemented in a few lines in Pregel. Pregel has become another popular tool after MapReduce.

Similar to Pregel, GraphLab [7][8], PowerGraph [3] and Graph-Chi [6], provide alternative graph processing model. The main difference between Pregel and others is the synchronization model. For example, Pregel splits the processing into multiple supersteps and requires a global synchronization at the end of each superstep, while GraphLab does not have such requirement and it is completely asynchronous. GraphLab provides three different consistent models: full, edge, and vertex consistency.

Recently, some memory-based distributed processing platforms, such as Spark [18] and Trinity [12], are proposed as high-performance data analytical systems. They share some similar design philosophy with the Pregel-like systems. We can simulate Pregel on top of Spark or Trinity. However, to improve the efficiency, those systems maintain the graph structures or intermediate results in memory to reduce the I/O cost. Our technique can benefit those systems by avoiding the unnecessary iterations as in the Pregel system. Some recent works[16][13] also proposed the idea similar with super-vertex, but they differ from our vertex construction and computing processing model.

5 Conclusion

Pregel was recently proposed as a processing engine for large graphs. In Pregel, the algorithm is invoked iteratively and each iteration is called a superstep. All vertices need to reach the same status at the end of each superstep, which incurs high synchronization cost. To address this problem, we have extended Pregel to our new framework P++. P++ introduces a new data structure, the *super-vertex*, and a new processing interface, *internalCompute*. The super-vertex represents a set of graph vertices and is hosted by a single cluster node. All computations within the super-vertex can be processed locally within each superstep. and hence, the algorithm requires fewer supersteps to complete. Consequently, the synchronization cost is greatly reduced. P++ is completely compatible with the Pregel in that all Pregel codes can run in the P++ without modification in order to gain performance improvement. We used shortest path algorithm to illustrate the benefit of P++. P++ is evaluated using real datasets and the results demonstrate its superior performance.

Acknowledgement. This research has been supported by The National Key Technology R&D Program of the Ministry of Science and Technology of China (Grant No. 2013BAG06B01) and the National Science Foundation of China (NSFC Grant 61202047).

References

1. Bahmani, B., Chakrabarti, K., Xin, D.: Fast personalized pagerank on mapreduce. In: SIGMOD (2011)
2. Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. In: OSDI (2004)
3. Gonzalez, J.E., Low, Y., Gu, H., Bickson, D., Guestrin, C.: Powergraph: Distributed graph-parallel computation on natural graphs. In: OSDI (2012)
4. Hewitt, C., Bishop, P., Steiger, R.: A universal modular actor formalism for artificial intelligence. In: IJCAI (1973)
5. Kwak, H., Lee, C., Park, H., Moon, S.: What is Twitter, a social network or a news media? In: WWW (2010)
6. Kyrola, A., Blelloch, G., Guestrin, C.: Graphchi: Large-scale graph computation on just a pc. In: OSDI (2012)
7. Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., Hellerstein, J.M.: Graphlab: A new framework for parallel machine learning. In: UAI (2010)
8. Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., Hellerstein, J.M.: Distributed graphlab: A framework for machine learning in the cloud. PVLDB 5(8) (2012)
9. Malewicz, G., Austern, M.H., Bik, A.J.C., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: a system for large-scale graph processing. In: SIGMOD (2010)
10. Salihoglu, S., Widom, J.: Gps: A graph processing system. In: Technical Report, Stanford (2012)
11. Schloegel, K., Karypis, G., Kumar, V.: Parallel multilevel algorithms for multi-constraint graph partitioning. In: Bode, A., Ludwig, T., Karl, W.C., Wismüller, R. (eds.) Euro-Par 2000. LNCS, vol. 1900, p. 296. Springer, Heidelberg (2000)
12. Shao, B., Wang, H., Li, Y.: Trinity: A distributed graph engine on a memory cloud. In: SIGMOD (2013)
13. Tian, Y., Balmin, A., Corsten, S.A., Tatikonda, S., McPherson, J.: From "think like a vertex" to "think like a graph". PVLDB 7(3), 193–204 (2013)
14. Valiant, L.G.: A bridging model for parallel computation. Communications of the ACM 33(8) (1990)
15. Wang, Y., DeWitt, D.J.: Computing pagerank in a distributed internet search engine system. In: VLDB (2004)
16. Xie, W., Wang, G., Bindel, D., Demers, A.J., Gehrke, J.: Fast iterative graph computation with block updates. PVLDB 6(14), 2014–2025 (2013)
17. Yang, S., Yan, X., Zong, B., Khan, A.: Towards effective partition management for large graphs. In: SIGMOD (2012)
18. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: Cluster computing with working sets. In: HotCloud (2010)