# Kylin: An Efficient and Scalable Graph Data Processing System

Li-Yung Ho, Tsung-Han Li
*Department of Computer Science and Information Engineering*
*National Taiwan University*
*Taipei, Taiwan*
*Email: {d96025,r00922074}@csie.ntu.edu.tw*

Jan-Jan Wu
*Institute of Information Science*
*Research Center for Information Technology Innovation*
*Academia Sinica*
*Taipei, Taiwan*
*Email: wuj@iis.sinica.edu.tw*

Pangfeng Liu
*Department of Computer Science and Information Engineering,*
*Graduate Intitute of Networking and Multimedia,*
*National Taiwan University*
*Taipei, Taiwan*
*Email: pangfeng@csie.ntu.edu.tw*

***Abstract*—We introduce *Kylin*, an efficient and scalable graph data processing system. Kylin is based on bulk synchronization processing(BSP) model to process graph data. Although there have been some BSP-based graph processing systems, Kylin is different from these systems in two-fold. First, Kylin cooperates with HBase to achieve scalable data manipulation. Second, We propose three techniques to optimize the performance of Kylin. The proposed techniques are pull messaging, lazy vertex loading and vertex-weighted partitioning. We demonstrate Kylin outperforms other BSP-based systems, i.e. Hama and Giraph, in the experiments.**

***Keywords*-graph data processing, graph data partition, load balancing, pull messaging, dynamic loading**

## I. Introduction

*Bulk synchronous parallel (BSP)* is a common programming model for graph data processing. There are many open source implementations like Giraph [1] and Hama [2]. Although it is popular, there are several problems in current systems. First, the synchronization phase becomes the performance bottleneck if the massive message passing among the workers is not handled carefully. Second, existing systems use hashing to distribute graph data to the distributed storage, and hence suffers lack of data locality during data processing. Finally, existing BSP-based systems use Hadoop file system (HDFS) as their storage system. HDFS is a file-based system and does not provide any data scheme and API for data management. As a result, users have to implement various *input format* to read the records from files for processing. Moreover, HDFS does not support indexing mechanism so we can not retrieve a specified set of data efficiently from HDFS.

To address the above problems, we develop *Kylin*, an optimized BSP data processing system built on HBase. The reason for choosing Hbase as our storage system is that HBase provides very useful data management features, including indexing for efficient data retrieval and structured data scheme. These features substantially alleviate our development effort in building the Kylin system.

We also devise three optimization techniques to improve data partitioning, message passing and memory management in BSP data processing: (1) *vertex-weighted partitioning*, which achieve data locality while balancing the load of each partition, (2) *pull messaging*, which reduces the interprocessor communication traffic by eliminating unnecessary message passing, and (3) *lazy vertex loading*, which saves memory usage by avoiding loading unneeded vertices into memory.

There are four contributions of this paper. First, we develop *Kylin*, an optimized BSP framework in Hadoop/HBase. To the best of our knowledge, our work is the first effort to develop the BSP framework on HBase. Second,we propose three optimization techniques, *vertex-weighted partitioning*, *Pull messaging* and *lazy vertex loading*, to improve the performance of BSP data processing. Third, we develop a comprehensive benchmark suite to evaluate the existing BSP systems and Kylin. Fourth, we conduct extensive experiments to evaluate the performance of existing BSP systems and Kylin. Kylin outperforms existing systems from 1.2x to 5x for different applications.

The rest of the paper is organized as follows. Section II discusses some related works and highlight the differences between our work and existing work. Section III describes the software architecture of the Kylin BSP framework we develop on HBase. Section IV presents the three optimization techniques we propose. Section V reports our experiment results, and Section VI gives some concluding remarks.

## II. Related Works

Pregel [3] is a framework based on bulk synchronization parallel model(BSP). Pregel is designed to process large scale graph data. Pregel employs a *master-slave* architecture. In this architecture a master orchestrates a set of worker to complete a computation job. The job is divided into tasks and each worker process some tasks. The job completes when all tasks finish. In the system initialization phase of Pregel, every worker registers itself to the master, therefore, the master knows the IP and the port number of every worker. Then, to process a job the input files are divided into *splits* and the master assigns each split to a worker. The worker reads graph information from the split and build the *vertices*.

A vertex is a computation unit in Pregel. User defines the *compute* function on the vertex for data processing. A job in Pregel consists of several *supersteps*. In each superstep, all vertices perform the compute function independently. Once all vertices finish, they synchronize and communicate with each other to exchange data. Another superstep begins after this synchronization. The job finish as soon as all vertices vote to halt. Giraph [1] and Hama [2] are an open source implementation of Pregel.

Although Kylin is also based on the BSP model, its implementation strategy and optimization techniques distinguish Kylin from other existing systems. We highly optimize the BSP engine in Kylin to achieve better performance, in terms of data partitioning, message passing and memory management.

## III. ARCHITECTURE

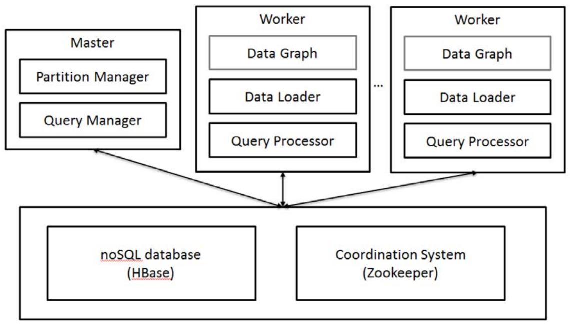


Figure 1. The architecture of Kylin

Figure 1 depicts the software architecture of Kylin. Graph data stored in the NoSQL database will be fed into the partition manager, which performs vertex-oriented disjoint partitioning of the graph. The resulting partition information will also be stored in the NoSQL database. Kylin employs a master-slave architecture for easy management of superstep executions as suggested in Pregel [3]. The workflow of Kylin is described as follows.

Step 1: A client sends a request to the master. The master generates worker tasks from the request and then distributes the tasks to the workers based on a pre-specified partitioning strategy.

Step 2: After receiving a task from the master, a worker loads its portion of the graph assigned by the master from the NoSQL database.

Step 3: The master and the workers enter the Bulk Synchronous Processing supersteps as described in the following. In each superstep, each worker iterates over all vertices in its local partition and executes an user defined compute function. The workers exchange messages to move graph data that are accessed across partitions. At the end of each superstep, the master coordinates the synchronization among the workers before entering the next superstep. The iteration of the supersteps terminates when all vertices become non-active. Finally, the workers store the computation result back to the NoSQL database.

In the following subsections, we describe each module in more details.

### A. Partition Manager

The partition manager is responsible of partitioning the graph and storing the partition information into the NoSQL database. Each worker consults the partition information from the NoSQL database and loads graph data according to the partition information. If the number of workers change either due to machine failure or addition of new workers, the partition manager re-partitions the graph to re-balance the workload.

### B. Query Manager

The query manager serves multiple purposes in our architecture: 1) receiving requests from clients, 2) assigning tasks to query processors with parameters specified by the client and 3) coordinating executions of supersteps. The query manager and query processor communicate via a coordination system to perform superstep synchronization. Employing a coordination system makes it easier to synchronize between supersteps and also ease the burden of the master.

### C. Data Graph and Data Loader

A client can provide parameters to the master to specify whether the computation would require loading the whole graph or only a particular sub-graph. The data loader can then load a portion of the data graph determined by both the partition information and client provided parameters. The graph data stored in the NoSQL database is structured as an adjacency list. A key-value pair in the NoSQL database represents a vertex and its neighbor.

### D. Query Processor

The query processor is the place where vertex compute program is executed. Each query processor keeps track of a queue of active vertices, which was gradually emptied as each vertex program execution completes. It also send messages and pull requests to other workers for the computation on the vertices at the partition boundaries. After completing execution of its partition, the query processor consults the coordination system to get current state from master in order to synchronize with other workers. We will discuss the pull requests and dynamic pull model in the next section.

We use HBase as the noSQL database in our implementation. However, other noSQL database could also be used as the backend store. We choose HBase because it comes with a mature coordination system, Zookeeper.

## IV. Optimizations

In this section, we present three optimization techniques we devise for efficient bulk synchronous model, including *pull messaging*, *lazy vertex loading*, and *vertex-weighted partitioning*.

### A. Pull Messaging

Kylin uses a *pull model* for message passing; that is, a vertex *pulls* its neighbors' data when that vertex needs to perform computation. Compared to the push model which is used in Giraph and Hama, the pull model can eliminate significant amount of unnecessary messages. There are two sources of unnecessary message passing. The first is that the sender may send duplicated messages to the receivers. A message is duplicated because it has the same content as the last message from the same sender. A vertex will continually send duplicated messages if that vertex's state does not change significantly. For example, in the PageRank application, even after the page rank of a vertex converges, that vertex will still send duplicated messages to its neighbors in subsequent supersteps. Another form of unnecessary message passing occurs as the overhead of broadcast operation. The broadcast operation is unnecessary for vertices residing in the same machine because the neighboring data can be accessed in memory.

For broadcast messages transmitted across machines, pull model can also help reduce the message size from $O(E)$ to $O(V)$ as illustrated in figure 2.

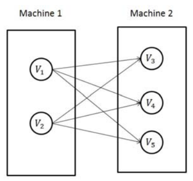


Figure 2. For machine 2, vertex 3,4, 5 need information from vertex 1 and vertex 2 on machine 1. The broadcast operation would result in 6 messages. Whereas the pull model requires only 2 messages.

In Kylin, a set of *active* vertices decide whether to perform computation or not according to its own change of state in every superstep. An active vertex that will perform computation needs to pull all it neighbors data before the computation. For efficient data processing, we pull the neighbors data in batches so that we can perform data transmission and computation in a pipelined manner as described below. We divide the active vertices into several subsets. In each pipeline stage, we retrieve the neighbors data of a subset active vertices. An active vertex that has all its neighbors data can start the computation. At the same time, we can retrieve the next batch of neighbors data for another subset active vertices. Such pipeline processing can overlap computation and communication effectively to hide the communication overhead.

In order to minimize the total execution time of such pipeline processing, we need to solve an important problem: how to find a set of active vertices to compute in a pipeline stage? The number of selected active vertices should be as large as possible so that we can finish the computation for more vertices at a time. On the other hand, the retrieved neighbors data cannot exceed certain amount due to network bandwidth constraint.

We formulate this problem as a *maximum active subset problem*, which is stated as follows. Given a set of active vertices and their corresponding neighbors, find a maximum subset of active vertices such that the number of neighbors of that subset is within a given bound. Note that we exclude active vertices in the neighbors, that is, if there is an active vertex $u$ whose neighbor $v$ is also an active vertex, we remove $v$ from the neighbor set, because we do not need to transmit the data of $v$.

*1) Problem definition:* We model the relations between active vertices and their neighbors as a bipartite graph. Note that the edges in our model are undirected. Given a graph $G = (V, E)$ and a set of active vertices $V_1 \subseteq V$, we can construct a bipartite graph $G' = (V_1 \cup V_2, E')$, where $V_2$ represents the neighbors of the active vertices $V_1$ and $E'$ is a subset of $E$. $E'$ consists of edges between $V_1$ and $V_2$ in the original graph $G$. We define *maximum active subset problem* as follows.

*Definition 1:* Given a bipartite graph $G' = (V_1 \cup V_2, E')$, and two integers $k$ and $l$, find a subset $\Omega$ of $V_1$ of size at least $k$, such that the set of neighbors of $\Omega$, $\Gamma$, is at most of size $l$.
We next show that the *maximum active subset* problem is NP-hard.

*2) NP hardness:* We define the decision version of maximum active subset problem as follows: Given a bipartite graph $G' = (V_1 \cup V_2, E')$, and two integers $k$ and $l$, does there exist a set $\Omega \subseteq V_1$ with size at least $k$, such that $|\Gamma|$ is at most $l$, where $\Gamma$ is the set of neighbors of $\Omega$?

*Theorem 1:* The decision version of the maximum active subset problem is NP-hard.

*Proof:* We reduce the *maximum independent set problem* [4] to the maximum active subset problem. We first show the if direction; that is, if we have a positive instance of maximum independent set, we have a positive instance of maximum active subset. Given an instance of maximum independent set problem, we have a graph $G = (V, E)$ and an independent set $\mathbb{I}$ with size at least $n$. We construct a bipartite graph $G' = (V \cup V, E')$ where $E' = \{(u, v), (v, u) | \text{if } u, v \text{ has an edge in } E\}$. We let the active subset $\Omega$ in the left hand side of the bipartite graph to be $\mathbb{I}$ and the neighbor set $\Gamma$ in the right hand side to be $V - \mathbb{I}$. By setting $k = n$ and $l = |V| - n$, we have a positive instance of maximum active subset problem, since we find a set $\Omega$ with size at least $k$ and its neighbor set with size at most $l$.

For the only if direction, suppose we have a bipartite graph $G' = (X \cup Y, E')$ and two subsets $\Omega \subseteq X$, $\Gamma \subseteq Y$ which satisfy $|\Omega| = k$ and $|\Gamma| = l$. Furthermore, $\Omega$ is the active set and $\Omega$ is the corresponding neighbors. Note that $\Omega \cap \Gamma =$. We then have an independent set $\mathbb{I} = \Omega$ of graph $G'' = (\Omega \cup \Gamma, E'')$ where $E'' = \{(u, v)|$ if $(u, v) \in E', \forall u \in \Omega, \forall v \in \Gamma\}$ ■

*3) Approximation algorithm:* Since the maximum active subset problem is computationally intractable (NP-hard), we devise an approximation algorithm to solve this problem. Our approximation algorithm is based on the optimal algorithm that finds the maximum independent set on a bipartite graph [5]. Algorithm 1 shows the approximation algorithm. We describe the procedures of Algorithm 1 in details. The input is a bipartite graph $G = (V_1 \cup V_2, E)$. We first find the maximum independent set of the given bipartite graph(*FindMaxIndependent* function in line 1). This can be done efficiently according to König's theorem [6]. The output of the function *FindMaxIndependent* is a maximum independent set $\mathbb{I}$. We then adjust $\mathbb{I}$ to obtain the final answer $\Omega$. $\mathbb{I}$ is composed of two sets, $L$ and $R$, which are the subsets of $V_1$ and $V_2$ respectively. We initially set $\Omega$ to be $L$(line 3). We then extend $\Omega$ by the following procedures(line 4-9). For each vertex $y$ in $R$, we choose an arbitrary neighbor $x$ of $y$ and add $x$ into $\Omega$ if $x \notin \Omega$. If $x$ is already in $\Omega$, we just skip $y$ and choose the next member of $R$. After iterating all members of $R$, we obtain the approximated maximum active subset $\Omega$.

**Algorithm 1** Find approximated active subset

```
Require: A bipartite graph $G = (V_1 \cup V_2, E)$
Ensure: A active subset $\Omega \subseteq V_1$
 1: $\mathbb{I} = FindMaxIndependentSet(G)$
 2: $L \cup R = \mathbb{I}$ where $L \subseteq V_1$ and $R \subseteq V_2$
 3: $\Omega = L$
 4: for all $y \in R$ do
 5:     $x = SelectOneNeighbor(y)$
 6:     if $x \notin \Omega$ then
 7:         $\Omega = \Omega \cup x$
 8:     end if
 9: end for
10: return $\Omega$
```

Theorem 2 proves the approximation bound of Algorithm 1.

*Theorem 2:* Algorithm 1 is a $\frac{1}{d}$-approximation algorithm for the maximum active subset problem, where $d$ is the average degree of a vertex in the bipartite graph.

*Proof:* Given a bipartite graph $G = (V_1 \cup V_2, E)$, suppose the maximum independent set of $G$ is $\mathbb{I}$ and $|\mathbb{I}| = p^*$. Furthermore, assume that the optimal solution of maximum active subset problem is $\Omega$ and $|\Omega| = p^\dagger$. We have $p^* \geq p^\dagger$, since maximum active subset is a special case of maximum independent set by constraining all vertices in $V_1$. Suppose the output of Algorithm 1 is $\Omega'$ and $|\Omega'| = p$. We have $p \geq \frac{1}{d}p^*$, because in line 5 of Algorithm 1, in the worst case, on average, there are $d$ vertices of $\mathbb{I}$ that choose the same vertex $x$. Therefore, the size of $\mathbb{I}$ will shrink to $\frac{1}{d}$. As a result, we have $p \geq \frac{1}{d}p^* \geq \frac{1}{n}p^\dagger$. That is, the size of $\Omega'$ obtained by Algorithm 1 is at least $\frac{1}{d}$ of the optimal solution to the maximum active subset problem. ■

## B. Lazy Vertex Loading

Kylin provides a *lazy vertex loading* strategy to load data into memory. When using lazy loading mode, some specified vertices will be loaded into memory initially. These initially loaded vertices are specified by users. Then, in the following super steps, only the vertices which receive messages to compute will be loaded into memory. As a result, we eliminate memory loading of vertices that do not need to perform computation or do not receive any messages (which means these vertices will not be reached by the initial vertices), so that we can utilize the memory resource more efficiently. Note that for applications that need the whole graph to compute, e.g. Max value propagation [3] or PageRank [7], all graph data will be loaded into memory.

Although lazy vertex loading can avoid loading unneeded vertices into memory, it may cause significant performance degradation if we do not manage the I/O carefully. Due to the random access nature of graph processing, lazy vertex loading may cause random disk accessing while loading the vertices into memory. To address this problem, we take advantage of the data layout management in HBase to make lazy vertices loading efficient. We use vertex ID as the key to store graph data in HBase. In the lazy vertex loading mode, the ID of a vertex will be inserted into a list if that vertex is not loaded in the memory and it has messages to process. In the synchronization phase between each superstep, we first sort the list storing the vertices ID in ascending order, and load those vertices accordingly by HBase APIs. Since HBase stores data with contiguous key in a HTable, loading vertices in the ascending order of vertex ID will result in sequential disk accesses to a HTable, which in turn significantly improves the I/O performance of lazy vertex loading.

## C. Vertex-weighted Partitioning

We formulate the vertex-weighted partitioning problem as follows. Given a graph $G = (V, E)$, we define a quantity *vertex weight* $\omega_v$ of a vertex $v \in V$ to be the number of edges adjacent to $v$. A partition $P$ is a subset of $V$. An *$n$ partition* $\mathbb{P}$ of $G$ is a set of partitions: $\mathbb{P} = \{P_i | i = 1, n\}$, and $P_i \cap P_j = \forall i \neq j$ and $V = \cup_i P_i$. A *cut-edge* is an edge whose end nodes belong to two different partitions. That is , for a cut-edge $e = (u, v)$, we have $u \in P_i$ and $v \in P_j, i \neq j$. We define $\theta_{ij}$ to be the set of cut-edges between two partition $P_i$ and $P_j$.

A vertex-weighted partitioning is a n-partition $\mathbb{PV}$ of $V$ satisfying the following two conditions.

- $\sum_{v \in P_i} \omega_v = \sum_{u \in P_j} \omega_u \forall i \neq j$
- $\sum_{i,j} |\theta_{ij}|$ is minimized $, \forall i, j \in \{1, \cdots, n\}, i \neq j$

The first condition means the weight of each partition is *balanced*. The second condition is to minimize the total number of cut-edges. The objective of vertex-weighted partitioning problem is to minimize number of cut-edges between partitions while keeping the summation of all in-degrees in a partition balanced. This graph partitioning problem is a well-known NP-complete problem [8]. We use an open source graph partitioner, METIS [8], to implement our vertex-weighted partitioning.

## V. Experiments

We conduct extensive experiments to evaluate the performance of Kylin. We first examine the benefits of the proposed optimization techniques, and then we combine all optimizations to obtain the overall performance. We compare overall performance of Kylin with two existing BSP systems: Apache Hama and Apache Giraph.

### A. Setup

The experimental environment is a 16-node cluster. Eight machines of them are Intel Xeon E5504 machines and each has 13 gigabyte RAM. The other eight machines are Intel Xeon E5520 machines and each has 23 gigabyte RAM. We run each application 10 times and use the average execution time as the performance metric. The version of Hama we used is 0.6.0 and the version of Giraph is 1.0.0. We use Hadoop 1.0.2 to be the based system of Hama and Giraph. Kylin is implemented on HBase 0.94.6-cdh4.3.0. The graph partitioner METIS is 5.1.0.

We collected four real world social networks data to be the input graph data of of our benchmark suite. The four social networks are: Youtube, Flicker, LiveJournal and Orkut. The four graph data are published by Mislove [9] in the anonymous form. In the four graph data, a vertex represents a person and the edges stand for the social relationships among the persons. Table I summarize the number of nodes and edges of the four social graphs.

| Social Networks | Nodes (millions) | Edges (millions) |
|---|---|---|
| Orkut | 3.07 | 117.26 |
| Flicker | 1.86 | 15.97 |
| LiveJournal | 5.28 | 49.4 |
| YouTube | 1.16 | 3.01 |

Table I
THE NUMBERS OF NODES AND EDGES OF FOUR SOCIAL NETWORKS

To evaluate the performance of Kylin, we collect and develop a set of representative graph algorithms to be the benchmark suite. Most of the algorithms are suggested by [3] and some of them are used in social network analysis [10], [11]. The benchmark suite includes the following seven applications: Maxvalue, N-neighbors, PageRank, Bipartite matching, Single source shortest path, Inference spreading, Label propagation.

### B. Pull model

In this experiment, we demonstrate the efficiency of pull messaging on PageRank application. Table II compares push and pull mode on Kylin for different social networks. It shows that in small graph e.g. YouTube, we can not benefit from pull messaging. However, for large graph like Orkut, we can achieve more than 2x speedup.

| Networks | Push(seconds) | Pull(seconds) |
|---|---|---|
| YouTube | 12.213 | 22.053 |
| Flicker | 128.792 | 95.934 |
| LiveJournal | 519.34 | 294.772 |
| Orkut | 1483.359 | 509.723 |

Table II
PUSH AND PULL MODE OF PageRank

### C. Lazy Vertex Loading

We evaluate the efficiency of lazy vertex loading for Nneighbors application. The partitioning strategy is vertex-weighted partitioning. Table III shows the results of running lazy vertex loading mode and normal mode. Normal mode loads the whole graph initially. The performance gain is significant if the subgraph involved in the computation is related small to the whole graph, e.g. the Orkut case. However, the performance becomes slightly worse for small graphs because of the extra overhead of data loading. The overhead of data loading in each superstep becomes significant especially for data with small size.

| Networks | Whole(seconds) | Lazy(seconds) |
|---|---|---|
| YouTube | 3.581 | 5.141 |
| Flicker | 22.702 | 26.135 |
| LiveJournal | 71.008 | 61.124 |
| Orkut | 175.784 | 28.133 |

Table III
WHOLE GRAPH AND LAZY VERTEX LOADING OF Nneighbors

### D. Partitioning strategies

We compare three different partitioning strategies in terms of number of cut-edges, number of nodes and number of edges of each partition. The compared strategies are hash partitioning, node-based partitioning and vertex-weighted partitioning.

Figure 3 shows the performance of each application on Kylin with different partitioning strategies for Orkut. It demonstrates that vertex-weighted outperforms node-based in most of cases. When comparing with Hash, vertex-weighted partitioning has slightly worse performance for applications whose runtime are less than 200 seconds. This

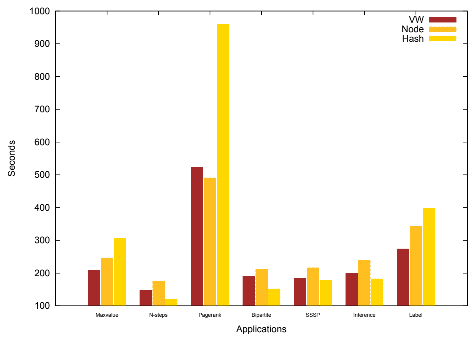
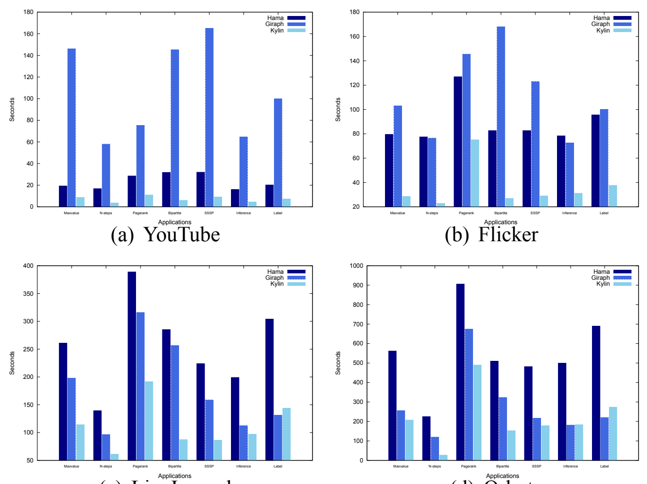


Figure 3. Different partition strategies for various applications on Kylin

is because for small applications, data transmission is not the bottleneck, so the balance of local computation becomes the most important factor of performance. Since Hash can achieve the most balanced load, it outperforms vertex-weighted partitioning. On the other hand, for applications suffer heavy data traffics, e.g. PageRank, vertex-weighted outperforms Hash significantly. This is because vertex-weighted achieves better data locality than Hash strategy and prevents remote message passing. Meanwhile, vertex-weighted also achieves load balancing by METIS.

## E. Overall performance



(a) YouTube (b) Flicker (c) LiveJournal (d) Orkut

Figure 4. Overall performances on the four social networks

We combine the optimizations mentioned above and conduct experiments to examine the overall performance of Kylin. Figure 4 illustrates the overall performance of Hama. Giraph and Kylin on various social networks. The x-axis represents different kinds of applications and the y-axis represents the job execution time in seconds in each subfigure. We first observe that in small social network, e.g. Youtube, Hama outperforms Giraph significantly. On the other hand, Giraph is more efficient than Hama to process large graphs e.g. Orkut. Kylin is efficient in either large or small graphs. For small graph, Kylin achieves 2x-5x performance improvement comparing with Hama. For large graph, Kylin outperforms Giraph by 1.2x-2x. Note that we take the best performance of each application by applying suitable optimizations. For example, we apply lazy vertex loading for Nneighbors application, but for PageRank. we load all the vertices initially.

## VI. Conclusion

We propose Kylin, an efficient and scalable BSP system for graph data processing. We optimize Kylin by three techniques: pull messaging, lazy vertex loading and vertex-weighted partitioning. We compare Kylin with two existing BSP system, Hama and Giraph. With the proposed optimizations, Kylin outperforms Hama and Giraph up to 5x and 2x performance improvement respectively.

## Acknowledgment

This work is supported by National Science Council of Taiwan under grant number NSC101-2221-E-001-004-MY3.

## References

[1] "Giraph," http://giraph.apache.org/.

[2] "Hama," http://hama.apache.org/.

[3] 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 *Proceedings of the 2010 international conference on Management of data*.

[4] C. D. Godsil and G. Royle, *Algebraic graph theory*. Springer New York, 2001.

[5] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network flows: theory, algorithms, and applications*, 1993.

[6] K. Dénse, "Gráfok és mátrixpk," *Matematikai és Fizikai Lapok*, vol. 38.

[7] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Technical Report, 1999.

[8] G. Karypis and V. Kumar, "Multilevel algorithms for multi-constraint graph partitioning," in *Supercomputing, 1998. SC98. IEEE/ACM Conference on*.

[9] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee, "Measurement and Analysis of Online Social Networks," in *Proceedings of the 5th ACM/Usenix Internet Measurement Conference (IMC'07)*.

[10] W. Chen, Y. Yuan, and L. Zhang, "Scalable influence maximization in social networks under the linear threshold model," in *Proceedings of the 2010 IEEE International Conference on Data Mining*.

[11] D. Kempe, J. Kleinberg, and E. Tardos, "Maximizing the spread of influence through a social network," in *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*.