Blogel: A Block-Centric Framework for Distributed Computation on Real-World Graphs

Da Yan\textsuperscript{1,}, James Cheng\textsuperscript{2,}, Yi Lu\textsuperscript{3,}, Wilfred Ng\textsuperscript{4,}\textsuperscript{*}

\textsuperscript{1}Department of Computer Science and Engineering, The Chinese University of Hong Kong
\{yanda, jcheng, ylu\}@cse.cuhk.edu.hk
\textsuperscript{2}Department of Computer Science and Engineering, The Hong Kong University of Science and Technology
\textsuperscript{3}wilfred@cse.ust.hk

ABSTRACT

The rapid growth in the volume of many real-world graphs (e.g., social networks, web graphs, and spatial networks) has led to the development of various vertex-centric distributed graph computing systems in recent years. However, real-world graphs from different domains have very different characteristics, which often create bottlenecks in vertex-centric parallel graph computation. We identify three such important characteristics from a wide spectrum of real-world graphs, namely (1) skewed degree distribution, (2) large diameter, and (3) relatively high density. Among them, only (1) has been studied by existing systems, but many real-world power-law graphs also exhibit the characteristics of (2) and (3). In this paper, we propose a block-centric framework, called Blogel, which naturally handles all the three adverse graph characteristics. Blogel programmers may think like a block and develop efficient algorithms for various graph problems. We propose parallel algorithms to partition an arbitrary graph into blocks efficiently, and block-centric programs are then run over these blocks. Our experiments on large real-world graphs verified that Blogel is able to achieve orders of magnitude performance improvements over the state-of-the-art distributed graph computing systems.

1. INTRODUCTION

Due to the growing need to deal with massive graphs in various analytic graph and graph mining applications, many distributed graph computing systems have emerged in recent years, including Pregel [11], GraphLab [10], PowerGraph [4], Giraph [1], GPS [15], and Mizan [8]. Most of these systems adopt the vertex-centric model proposed in [11], which promotes the philosophy of “thinking like a vertex” that makes the design of distributed graph algorithms more natural and easier. However, the vertex-centric model has largely ignored the characteristics of real-world graphs in its design and can hence suffer from severe performance problems.

We investigated a broad spectrum of real-world graphs and identified three characteristics of large real-world graphs, namely (1) skewed degree distribution (common for power-law and scale free graphs such as social networks and web graphs), (2) relatively high density (common for social networks, mobile phone networks, SMS networks, some web graphs, and the cores of most large graphs), and (3) large diameter (common for road networks, terrain graphs, and some large web graphs). These three characteristics are particularly adverse to vertex-centric parallelization as they are often the major cause(s) to one or more of the following three performance bottlenecks: skewed workload distribution, heavy message passing, and impractically many rounds of computation.

Let us first examine the performance bottleneck created by skewed degree distribution. The vertex-centric model assigns each vertex together with its adjacency list to a machine, but neglects the difference in the number of neighbors among different vertices. As a result, for graphs with skewed degree distribution, it creates an unbalanced workload distribution that leads to a long elapsed running time due to waiting for the last worker to complete its job. For example, the maximum degree of the BTC RDF graph used in our experiments is 1,637,619, and thus a machine holding such a high-degree vertex needs to process many incoming messages and send out many outgoing messages to its neighbors, causing imbalanced workload among different machines.

Some existing systems proposed techniques for better load balancing [4, 15], but they do not reduce the overall workload. However, for many real-world graphs including power-law graphs such as social networks and mobile phone networks, the average vertex degree is large. Also, most large real-world graphs have a high-density core (e.g., the $k$-core [16] and $k$-truss [19] of these graphs). Higher density implies heavier message passing for vertex-centric systems. We show that heavy communication workload due to high density can also be eliminated by our new computing model.

For processing graphs with a large diameter $\delta$, the message (or neighbor) propagation paradigm of the vertex-centric model often leads to algorithms that require $O(\delta)$ rounds (also called supersteps) of computation. For example, a single-source shortest path algorithm in [11] takes 10,789 supersteps on a USA road network. Apart from spatial networks, some large web graphs also have large diameters (from a few hundred to thousands). For example, the vertex-centric system in [14] takes 2,450 rounds for computing strongly connected components on a web graph.

To address the performance bottlenecks created by real-world graphs in vertex-centric systems, we propose a block-centric graph-parallel abstraction, called Blogel. Blogel is conceptually as simple as Pregel but works in coarser-grained graph units called blocks. Here, a block refers to a connected subgraph of the graph, and message exchanges occur among blocks.

Blogel naturally addresses the problem of skewed degree distribution since most (or all) of the neighbors of a high-degree vertex $v$ are inside $v$’s block, and they are processed by sequential in-memory algorithms without any message passing. Blogel also solves the heavy communication problem caused by high density,
since the neighbors of many vertices are now within the same block, and hence they do not need to send/receive messages to/from each other. Finally, Blogel effectively handles large-diameter graphs, since messages now propagate in the much larger unit of blocks instead of single vertices, and thus the number of rounds is significantly reduced. Also, since the number of blocks is usually orders of magnitude smaller than the number of vertices, the workload of a worker is significantly less than that of a vertex-centric algorithm.

A central issue to Blogel is whether we can partition an arbitrary input graph into blocks efficiently. We propose a graph Voronoi diagram based partitioner which is a fully distributed algorithm, while we also develop more effective partitioners for graphs with additional information available. Our experiments show that our partitioning algorithms are efficient and effective.

We present a user-friendly and flexible programming interface for Blogel, and illustrate that programming in Blogel is easy by designing algorithms for a number of classic graph problems in the Blogel framework. Our experiments on large real-world graphs with up to hundreds of millions of vertices and billions of edges, and with different characteristics, verify that our block-centric system is orders of magnitude faster than the state-of-the-art vertex-centric systems [1, 10, 4]. We also demonstrate that Blogel can effectively address the performance bottlenecks caused by the three adverse characteristics of real-world graphs.

The rest of the paper is organized as follows. Section 2 reviews related systems. Section 3 illustrates the merits of block-centric computing. Section 4 gives an overview of the Blogel framework, Section 5 presents Blogel’s programming interface, and Section 6 discusses algorithm design in Blogel. We describe our partitioners in Section 7 and present performance results in Section 8. Finally, we conclude our paper in Section 9.

2. BACKGROUND AND RELATED WORK

We first define some basic graph notations and discuss the storage of a graph in distributed systems.

**Notations.** Given an undirected graph \( G = (V, E) \), we denote the neighbors of a vertex \( v \in V \) by \( \Gamma(v) \); if \( G \) is directed, we denote the in-neighbors (out-neighbors) of \( v \) by \( \Gamma_{in}(v) \) (\( \Gamma_{out}(v) \)). Each \( v \in V \) has a unique integer ID, denoted by \( id(v) \). The diameter of \( G \) is denoted by \( \delta(G) \), or simply \( \delta \) when \( G \) is clear from the context.

**Graph storage.** We consider a shared-nothing architecture where data is stored in a distributed file system (DFS), such as Hadoop’s DFS (HDFS). We assume that a graph is stored as a distributed file in HDFS, where each line records a vertex and its adjacency list. A distributed graph computing system involves a cluster of \( k \) workers, where each worker \( w_i \) holds and processes a batch of vertices in its main memory. Here, “worker” is a general term for a computing unit, and a machine can have multiple workers in the form of threads/processes. A job is processed by a graph computing system in three phases: (1) **loading:** each worker loads a portion of vertices from HDFS into main-memory; the workers then exchange vertices through the network (by hashing over vertex ID) so that each worker \( w_i \) finally holds all and only those vertices assigned to \( w_i \); (2) **iterative computing:** in each iteration, each worker processes its own portion of vertices sequentially, while different workers run in parallel and exchange messages. (3) **dumping:** each worker writes the output of all its processed vertices to HDFS. Most existing graph-parallel systems follow this procedure.

**Pregel’s computing model.** Pregel [11] is designed based on the bulk synchronous parallel (BSP) model. It distributes vertices to workers, where each vertex is associated with its adjacency list. A program in Pregel implements the `compute()` function and proceeds in iterations (called **supersteps**). In each superstep, the program calls `compute()` for each active vertex. The `compute()` function performs the user-specified task for a vertex \( v \), such as processing \( v \)’s incoming messages (sent in the previous superstep), sending messages to other vertices (for the next superstep), and making \( v \) vote to halt. A halted vertex is reactivated if it receives a message in a subsequent superstep. The program terminates when all vertices vote to halt and there is no pending message for the next superstep.

Pregel also supports message combiner. For example, if there are \( k \) numerical messages in worker \( w_j \) to be sent to a vertex \( v \) in worker \( w_j \) and suppose that only the sum of the message values matters, then user can implement a `combine()` function to sum up the message values and deliver only the sum to \( v \) in \( w_j \), thus reducing the number of messages to be buffered and transmitted. Pregel also supports aggregator, which is useful for global communication. Each vertex can provide a value to an aggregator in `compute()` in a superstep. The system aggregates those values and makes the aggregated result available to all vertices in the next superstep.

Since Google’s Pregel is proprietary, many open-source systems have been developed based on Pregel’s computing model, such as Giraph [1] and GPS [15]. In particular, GPS uses a technique called large adjacency list partitioning to handle high-degree vertices.

**Vertex placement.** Vertex placement rules that are more sophisticated than hashing were studied in [17], which aims at minimizing the number of cross-worker edges while ensuring that workers hold approximately the same number of vertices. However, their method requires expensive preprocessing but the performance gain is limited (e.g., only a speed up of 18%–39% for running PageRank).

**Giraph++.** A recent system, Giraph++ [18], proposed a graph-centric programming paradigm that opens up the block structure to users. However, Giraph++’s programming paradigm is still vertex-centric since it does not allow a block to have its own states like a vertex. Instead, each block is treated as two sets of vertices, internal ones and boundary ones. As such, Giraph++ does not support block-level communication, i.e., message passing is still from vertices to vertices, rather than from blocks to blocks which is more efficient for solving some graph problems. For example, an algorithm for computing connected components is used to demonstrate Giraph++ programming in [18], in which vertex IDs are passed among internal vertices and boundary vertices. We implement an algorithm for the same problem in our block-centric framework in Section 3, and show that it is much simpler and more efficient to simply exchange block IDs between blocks directly. Moreover, since Giraph++ is Java-based, the intra-block computation inevitably incurs (de)serialization cost for each vertex accessed; in contrast, the intra-block computation in Blogel is simply a main-memory algorithm without any additional cost. Finally, Giraph++ extends METIS [7] for graph partitioning, which is an expensive method. In Section 8, we will show that graph partitioning and computing is much more efficient in Blogel than in Giraph++.

**GRACE.** Another recent system, GRACE [20], which works in a single-machine environment, also applies graph partitioning to improve performance. GRACE enhances vertex-centric computing with a scheduler that controls the order of vertex computation in a block. This can be regarded as a special case of block-centric computing, where the intra-block computing logic is totally defined by the scheduler. Although this model relieves users of the burden of implementing the intra-block computing logic, it is not as expressive as our block-centric paradigm. For example, GRACE does not allow a block to have its own states and does not support block-level communication. We remark that Blogel and GRACE have different focuses: GRACE attempts to improve main memory bandwidth utilization in a single-machine environment, while Blo-
gel aims to reduce the computation and communication workload in a distributed environment.

**GraphLab.** Unlike Pregel’s synchronous model and message passing paradigm, GraphLab [10] adopts a shared memory abstraction and supports asynchronous execution. It hides the communication details from programmers: when processing a vertex, users can directly access its own field as well as the fields of its adjacent edges and vertices. GraphLab maintains a global scheduler, and workers fetch vertices from the scheduler for processing, possibly adding the neighbors of these vertices into the scheduler. Asynchronous computing may decrease the workload for some algorithms but incurs extra cost due to data locking/unlocking. A recent version of GraphLab, called PowerGraph [4], partitions the graph by edges instead of by vertices, so that the edges of a high-degree vertex are handled by multiple workers. Accordingly, an edge-centric Gather-Apply-Scatter (GAS) computing model is used. However, GraphLab is less expressive than Pregel, since a vertex cannot communicate with a non-neighbor, and graph mutation is not supported.

### 3. Merits of Block-Centric Computing: A First Example

We first use an example to illustrate the main differences in the programming and performance between the block-centric model and vertex-centric model, by considering the Hash-Min algorithm of [13] for finding connected components (CCs). We will show the merits of the block-centric model for processing large real-world graphs with the three characteristics discussed in Section 1.

Given a CC $C$, we denote the set of vertices of $C$ by $V(C)$, and define the ID of a CC $C$ to be $cc(v) = \min\{id(u) : u \in V(C)\}$. Hash-Min computes $cc(v)$ for each vertex $v \in V$. The idea is to broadcast the smallest vertex ID seen so far by each vertex $v$, denoted by $\text{min}(v)$.

For the vertex-centric model, in step 1, each vertex $v$ sets $\text{min}(v)$ to be the smallest ID among $id(v)$ and $id(u)$ of all $u \in \Gamma(v)$, broadcasts $\text{min}(v)$ to all its neighbors, and votes to halt. In each later superstep, each vertex $v$ receives messages from its neighbors; let $\text{min}^*$ be the smallest ID received, if $\text{min}^* < \text{min}(v)$, $v$ sets $\text{min}(v) = \text{min}^*$ and broadcasts $\text{min}^*$ to its neighbors. All vertices vote to halt at the end of a superstep. When the process converges, $\text{min}(v) = cc(v)$ for all $v$.

Next we discuss the implementation of Hash-Min in the block-centric framework. Let us assume that vertices are already grouped into blocks by our partitioners (to be discussed in Section 7), which guarantee that vertices in a block are connected. Each vertex belongs to a unique block, and let $\text{block}(v)$ be the ID of the block that $v$ belongs to. Let $B$ be the set of blocks, and for each block $B \in B$, let $V(B)$ be the set of vertices of $B$, and $id(B)$ be the integer ID of $B$. We define $\Gamma(B) = \bigcup_{u \in V(B)} \{\text{block}(u) : u \in \Gamma(v)\}$. Thus, we obtain a block-level graph, $\mathbb{G} = (B, E)$, where $E = \{\{B_i, B_j\} : B_i, B_j \in \Gamma(B)\}$. We then simply run Hash-Min on $\mathbb{G}$ where blocks in $B$ broadcast the smallest block ID that they have seen. Similar to the vertex-centric algorithm, each block $B$ maintains a field $\text{min}(B)$, and when the algorithm converges, all vertices $v$ with the same $\text{min}(\text{block}(v))$ belong to one CC.

We now analyze why the block-centric model is more appealing than the vertex-centric model. First, we consider the processing of graphs with skewed degree distribution, where we compare $G$ with $\mathbb{G}$. Most real-world graphs with skewed degree distribution consists of a giant CC and many small CCs. In this case, our partitioner computes a set of roughly even-sized blocks from the giant CC, while each small CC forms a block. Let $b$ be the average number of vertices in a block and $d_{max}$ be the maximum vertex degree in $G$. The vertex-centric model works on $G$ and hence a high-degree vertex may send/receive $O(d_{max})$ messages each round, causing skewed workload among the workers. On the contrary, the block-centric model works on $\mathbb{G}$ and a high-degree vertex involves at most $O(n/b)$ messages each round, where $n$ is the number of vertices in the giant CC. For power-law graphs, $d_{max}$ can approach $n$ and $n/b$ can be a few orders of magnitude smaller than $d_{max}$ for a reasonable setting of $b$ (e.g., $b = 1000$).

In addition, as long as the number of blocks is sufficiently larger than the number of workers, the block-centric model can achieve a rather balanced workload with a simple greedy algorithm for block-to-worker assignment described in Section 4.

Figure 1 reports the overall performance (elapsed computing time, total number of messages sent, and total number of supersteps) of the vertex-centric and block-centric Hash-Min algorithms on three graphs: an RDF graph $BTC$, a social network $Friendster$, and a USA road network. The details of these graphs can be found in Section 8. Figure 2 reports the performance (running time and number of messages) of Hash-Min for each superstep on $BTC$ and $Friendster$.

$BTC$ has skewed degree distribution, as a few vertices have degree over a million but the average degree is only 4.69. The vertex-centric program is severely affected by the skewed workload due to high-degree vertices in the first few supersteps, and a large number of messages are passed as shown in Figure 2(a). The number of messages starts to drop significantly only in superstep 4 when the few extremely high-degree vertices become inactive. The subsequent supersteps involve less and less messages since the smallest vertex IDs have been seen by most vertices, and hence only a small fraction of the vertices remain active. However, it still takes 30 supersteps to complete the job. On the contrary, our block-centric program has balanced workload from the beginning and uses only 6 supersteps. Overall, as Figure 1 shows, the block-centric program uses 680 times less messages and 30 times less computing time.

Next, we discuss the processing of graphs with high density. Social networks and mobile phone networks often have relatively higher average degree than other real-world graphs. For example, Friendster has average degree of 55.06 while the USA road network has average degree of only 2.44 (see Figure 7), which implies that a vertex of Friendster can send/receive 22.57 times more messages than that of the USA road network. The total number of messages in each superstep of the vertex-centric Hash-Min is bounded by $O(|E|)$, while that of the block-centric Hash-Min is bounded by $O(|E|)$. Since $|E|$ is generally significantly smaller than $|E|$, the block-centric model is much cheaper.

As Figure 1 shows, the vertex-centric Hash-Min uses 372 times more messages than the block-centric Hash-Min on Friendster, resulting in 48 times longer computing time. Figure 2(b) further shows that the number of messages and the elapsed computing time per superstep of the vertex-centric program is significantly larger than that of the block-centric program. Note that the more message passing and longer computing time are not primarily due to skewed workload because the maximum degree in Friendster is only 5214.

Let $\delta(G)$ and $\delta(\mathbb{G})$ be the diameter of $G$ and $\mathbb{G}$. The vertex-centric Hash-Min takes $O(\delta(G))$ supersteps, while the block-centric Hash-Min takes only $O(\delta(\mathbb{G}))$ supersteps. For real-world graphs...
with a large diameter, such as road networks, condensing a large region (i.e., a block) of $G$ into a single vertex in $G$ allows distant points to be reached within a much smaller number of hops. For example, in the USA road network, there may be thousands of hops between Washington and Florida; but if we condense each state into a block, then the two states are just a few hops apart.

The adverse effect of large diameter on the vertex-centric model can be clearly demonstrated by the USA road network. As shown in Figure 1, the vertex-centric Hash-Min takes in 6,262 supersteps, while the block-centric Hash-Min runs on a block-level graph with a much smaller diameter and uses only 22 supersteps. The huge number of supersteps of the vertex-centric Hash-Min also results in 263 times longer computing time and 30,908 times more messages being passed than the block-centric algorithm.

4. SYSTEM OVERVIEW

We first give an overview of the Blogel framework. Blogel supports three types of jobs: (1) vertex-centric graph computing, where a worker is called a V-worker; (2) graph partitioning which groups vertices into blocks, where a worker is called a partitioner; (3) block-centric graph computing, where a worker is called a B-worker.

Figure 3 illustrates how the three types of workers operate. Figure 3(a) shows that each V-worker loads its own portion of vertices, performs vertex-centric graph computation, and then dumps the output of the processed vertices (marked as grey) to HDFS. Figure 3(b) shows block-centric graph computing, where we assume that every block contains two vertices. Specifically, the vertices are first grouped into blocks by the partitioners, which dump the constructed blocks to HDFS. These blocks are then loaded by the B-workers for block-centric computation.

Blocks and B-workers. In both vertex-centric graph computing and graph partitioning, when a vertex $v$ sends a message to a neighbor $u \in \Gamma(v)$, the worker to which the message is to be sent is identified by hashing $id(u)$. We now consider block-centric graph computing. Suppose that the vertex-to-block assignment and block-to-worker assignment are already given, we define the block of a vertex $v$ by $block(v)$ and the worker of a block $B$ by $worker(B)$. We also define $worker(v) = worker(block(v))$. Then, the ID of a vertex $v$ in a B-worker is now given by a triplet $trip(v) = (id(v), block(v), worker(v))$. Thus, a B-worker can obtain the worker and block of a vertex $v$ by checking its ID $trip(v)$.

Similar to a vertex in Pregel, a block in Blogel also has a compute() function. We use $B$-compute() and $V$-compute() to denote the compute() function of a block and a vertex, respectively. A block has access to all its vertices, and can send messages to any block $B$ or vertex $v$ as long as $worker(B)$ or $worker(v)$ is available. Each B-worker maintains two message buffers, one for exchanging vertex-level messages and the other for exchanging block-level messages. A block also has a state indicating whether it is active, and may vote to halt.
algorithm given in [5], which first sorts the jobs by cost in non-increasing order, and then scans through the sorted jobs and assigns each job to the worker with the smallest load.

The block-to-worker assignment is computed as follows. (1) To get the block sizes, each partitioner groups its vertices into blocks and sends the number of vertices in each block to the master. The master then aggregates the numbers sent by the partitioners to obtain the global number of vertices for each block. (2) The master then computes the block-to-worker assignment using the greedy algorithm described earlier, and broadcasts the assignment to each partitioner. (3) Each partitioner sets worker\(v\) for each of its vertex \(v\) according to the received block-to-worker assignment (note that block\(v\) is already computed).

Triplet ID of neighbors. So far we only compute trip\(v\) for each vertex \(v\). However, in block-centric computing, if a vertex \(v\) needs to send a message to a neighbor \(u \in \Gamma(v)\), it needs to first obtain the worker holding \(u\) from trip\(u\). Thus, we also compute \(\tilde{\Gamma}(v) = \{\text{trip}(u) : u \in \Gamma(v)\}\) for each vertex \(v\) as follows. Each worker \(w_i\) constructs a look-up table \(LT_{i,-}\) locally for every worker \(w_j\) in the cluster: for each vertex \(v\) in \(w_i\), and for each neighbor \(u \in \Gamma(v)\), trip\(v\) is added to \(LT_{i,-}\), where \(j = \text{hash}(id(u))\), i.e., \(u\) is on worker \(w_j\). Then, \(w_i\) sends \(LT_{i,-}\) to each \(w_j\), and each worker merges the received look-up tables into one look-up table. Now, a vertex \(u\) on worker \(w_j\) can find trip\(v\) for each neighbor \(v \in \Gamma(u)\) from the look-up table of \(w_j\) to construct \(\tilde{\Gamma}(u)\).

Till now, each partitioner \(w_i\) still holds only those vertices \(v\) with hash\(id(v)\) = \(i\). After \(\Gamma(v)\) is constructed for each vertex \(v\), the partitioners exchange vertices according to the block-to-worker assignment. Each partitioner \(w_i\) then drops its vertices to an HDFS file, which is later loaded by the corresponding B-worker \(w_i\) during block-centric computing. Each vertex \(v\) has an extra field, content\(v\), which keeps additional information such as edge weight and edge direction during data loading. It is used along with trip\(v\) and \(\tilde{\Gamma}(v)\), to format \(v\)'s output line during data dumping.

5. PROGRAMMING INTERFACE

Similar to Pregel, writing a Blogel program involves subclassing a group of predefined classes, with the template arguments of each base class properly specified. Figure 4(a) shows the base classes of Blogel. We illustrate their usage by showing how to implement the Hash-Min algorithm described in Section 3.

5.1 Vertex-Centric Interface

In the vertex-centric model, each vertex \(v\) maintains an integer ID id\(v\), an integer field cc\(v\) and a list of neighbors \(\Gamma(v)\). Thus, in the Vertex class, we specify IDType as integer, ValueType as a user-defined type for holding both cc\(v\) and \(\Gamma(v)\), and MsgType as integer since Hash-Min sends messages in the form of vertex ID. We then define a class CVCvertex to inherit this class, and implement the compute() function using the Hash-Min algorithm in Section 3. The Vertex class also has another template argument (not shown in Figure 4) for specifying the vertex-to-worker assignment function over IDType, for which a hash function is specified by default.

To run the Hash-Min job, we inherit the VWorker\(<\text{CCVertex}>\) class, and implement two functions: (1) load\(_\text{vertex}(\text{line})\), which specifies how to parse a line from the input file into a vertex object; (2) dump\(_\text{vertex}(\text{vertex}, \text{HDFSWriter})\), which specifies how to dump a vertex object to the output file. VWorker\(_\text{run}()\) is then called to start the job. BWorker and the partitioner classes also have these three functions, though the logic of run\(_()\) is different.

5.2 Block-Centric Interface

For the block-centric model, in the Vertex class, we specify IDType as a triplet for holding trip\(v\), ValueType as a list for holding \(\Gamma(v)\), while MsgType can be any type since the algorithm works in B-mode and there is no vertex-level message exchange. We then define a class CCVertex that inherits this class with an empty V-compute() function. In the Block class, we specify VertexType as CCVertex, BValueType as a block-level adjacency list (i.e., \(\Gamma(B)\)), and BBMsgType as integer since the algorithm sends messages in the form of block ID. We then define a class CCBlock that inherits this Block class, and implement the B-compute() function using the logic of block-centric Hash-Min. Finally, we inherits the BWorker\(<\text{CCBlock}>\) class, implement the vertex loading/dumping functions, and call run\(_()\) to start the job.

A Block object also has the following fields: (1) an integer block ID, (2) an array of the block's vertices, denoted by vertex, (3) and the number of vertices in the block, denoted by size.

A BWorker object also contains an array of blocks, denoted by block\(set\), and an array of vertices, denoted by vertex\(set\). As shown in Figure 4(b), the vertices in a B-worker's vertex\(set\) are grouped by blocks; and for each block \(B\) in the B-worker's block\(set\), \(B\)'s vertex\(set\) field is actually a pointer to the first vertex of \(B\)'s group in the B-worker's vertex\(set\). Thus, a block can access its vertices in B-compute\(_()\) as vertex\(_[0]\), vertex\(_[1]\), ..., vertex\(_[\text{size} - 1]\). A subclass of BWorker also needs to implement an additional function block\(_\text{init}()\), which specifies how to set the user-specified field BValue\(\text{Type}\) for each block. After a B-worker loads all its vertices into vertex\(set\), it scans vertex\(set\) to construct its block\(set\), where the fields vertex and size of each block are automatically set. Then, before the block-centric computing begins, block\(_\text{init}()\) is called to set the user-specified block field. For our block-centric Hash-Min algorithm, in block\(_\text{init}()\), each block \(B\) constructs \(\Gamma(B)\) from \(\Gamma(V)\) of all \(v \in V(B)\).

5.3 Global Interface

In Blogel, each worker is a computer process. We now look at a number of fields and functions that are global to each process. They can be accessed in both V-compute\(_()\) and B-compute\(_()\).

Message buffers and combiners. Each worker maintains two message buffers, one for exchanging vertex-level messages and the other for exchanging block-level messages. A combiner is associated with each message buffer, and it is not defined by default. To use a combiner, we inherit the Combiner class and implement the combiner\(_()\) function, which specifies the combiner logic. When a vertex or a block calls the send\(_\text{msg}(\text{target}, \text{msg})\) function of the message buffer, combine\(_()\) is called if the combiner is defined.
Aggregator. An aggregator works as follows. In each superstep, a vertex/block’s V-compute()B-compute() function may call aggregator(value), where value is of type AValueType. After a worker calls V-compute()B-compute() for all vertices/blocks, the aggregated object maintained by its local aggregator (of type Partial-Type) is sent to the master. When the master obtains the locally aggregated objects from all workers, it calls master_compute() to aggregate them to a global object of type FinalType. This global object is then broadcast to all the workers so that it is available to every vertex/block in the next superstep.

To define an aggregator, we subclass the Aggregator class to include a field recording the aggregated values, denoted by agg, and implement aggregator(value) and master_compute(). An object of this class is then assigned to the worker.

Blogel also supports other useful global functions such as graph mutation (which are user-defined functions to add/delete vertices/edges), and the terminate() function which can be called to terminate the job immediately. In addition, Blogel maintains the following global fields which are useful for implementing the computing logic: (1) the ID of the current superstep, which also indicates the number of supersteps executed so far; (2) the total number of vertices among all workers at the beginning of the current superstep, whose value may change due to graph mutation; (3) the total number of active vertices among all workers at the beginning of the current superstep.

6. APPLICATIONS

We apply Blogel to solve four classic graph problems: Connected Components (CCs), Single-Source Shortest Path (SSSP), Reachability, and PageRank. In Sections 3 and 5, we have discussed how Blogel computes CCs with the Hash-Min logic in both the vertex-centric model, and B-mode of the block-centric model. We now discuss the solutions to the other three problems in Blogel.

6.1 Single-Source Shortest Path

Given a graph \( G = (V,E) \), where each edge \((u,v) \in E\) has length \( \ell(u,v) \), and a source \( s \in V \), SSSP computes a shortest path from \( s \) to every other vertex \( v \in V \), denoted by \( SP(s,v) \).

Vertex-centric algorithm. We first discuss the vertex-centric algorithm, which is similar to Pregel’s SSSP algorithm [11]. Each vertex \( v \) has two fields: \((prev(v),dist(v))\) and \( \Gamma_{out}(v) \), where \( prev(v) \) is the vertex preceding \( v \) on \( SP(s,v) \) and \( dist(v) \) is the length of \( SP(s,v) \). Each \( u \in \Gamma_{out}(v) \) is associated with \( (v,u) \).

Initially, only \( s \) is active with \( dist(s)=0 \), while \( dist(v)=\infty \) for any other vertex \( v \). In superstep 1, \( s \) sends a message \((s, dist(s) + \ell(s,u))\) to each \( u \in \Gamma_{out}(s) \), and votes to halt. In superstep \( i \), if a vertex \( v \) receives messages \((u,d(w))\) from any of \( v \)'s in-neighbor \( w \), then \( v \) finds the in-neighbor \( w^* \) such that \( d(w^*) \) is the smallest among all \( d(w) \). If \( d(w^*) < dist(v) \), \( v \) updates \((prev(v), dist(v))=\Gamma_{out}(v) \), and sends a message \((v, dist(v) + \ell(v,u))\) to each out-neighbor \( u \in \Gamma_{out}(v) \). Finally, \( v \) votes to halt.

Let \( hop(s,v) \) be the number of hops of \( SP(s,v) \), and \( L = \max_{v \in V} hop(s,v) \). The vertex-centric algorithm runs for \( O(L) \) supersteps and in each superstep, at most one message is sent along each edge. Thus, the total workload is \( O(|E| + |V|) \).

Block-centric algorithm. Our block-centric solution operates in VB-mode. Each vertex maintains the same fields as in the vertex-centric algorithm, and blocks do not maintain any information. In each superstep, V-compute() is first executed for all vertices, where a vertex \( v \) finds \( w^* \) from the incoming messages as in the vertex-centric algorithm. However, now \( v \) votes to halt only if \( d(w^*) \geq dist(v) \). Otherwise, \( v \) updates \((prev(v), dist(v))=\Gamma_{out}(v) \) but stays active. Then, B-compute() is executed, where each block \( B \) collects all its active vertices \( v \) into a priority queue \( Q \) (with \( dist(v) \) as the key), and makes these vertices vote to halt. B-compute() then runs Dijkstra’s algorithm on \( B \) using \( Q \), which removes the vertex \( v \in Q \) with the smallest value of \( dist(v) \) from \( Q \) for processing each time. The out-neighbors \( u \in \Gamma(v) \) are updated as follows. For each \( u \in V(B) \), if \( dist(v)+\ell(v,u) < dist(u) \), we update \((prev(u), dist(u))\) to \((v, dist(v)+\ell(v,u))\), and insert \( u \) into \( Q \) with key \( dist(u) \) if \( u \notin Q \), or update \( dist(u) \) if \( u \) is already in \( Q \). For each \( u \notin V(B) \), a message \((v, dist(v)+\ell(v,u))\) is sent to \( u \). \( B \) votes to halt when \( Q \) becomes empty. In the next superstep, if a vertex \( u \) receives a message, \( u \) is activated along with its block, and the block-centric computation repeats.

Compared with the vertex-centric algorithm, this algorithm saves a significant amount of communication cost since there is no message passing among vertices within each block. In addition, messages propagate from \( s \) in the unit of blocks, and thus, the algorithm requires much less than supersteps than \( O(L) \).

For both the vertex-centric and block-centric algorithms, we apply a combiner as follows. Given a set of messages from a worker, \{\((w_1,d(w_1)), (w_2,d(w_2)), \ldots, (w_k,d(w_k))\)\}, to be sent to a vertex \( u \), the combiner combines them into a single message \((w^*,d(w^*))\) such that \( d(w^*) \) is the smallest among all \( d(w_i) \) for \( 1 \leq i \leq k \).

6.2 Reachability

Given a directed graph \( G = (V,E) \), a source vertex \( s \) and a destination vertex \( t \), the problem is to decide whether there is a directed path from \( s \) to \( t \) in \( G \). We can perform a bidirectional breadth-first search (BFS) from \( s \) and \( t \), and check whether the two BFSs meet at some vertex. We assign each vertex \( v \) a 2-bit field \( tag(v) \), where the first bit indicates whether \( s \) can reach \( v \) and the second bit indicates whether \( v \) can reach \( t \).

Vertex-centric algorithm. We first set \( tag(s) = 10 \), \( tag(t) = 01 \); and for any other vertex \( v \), \( tag(v) = 00 \). Only \( s \) and \( t \) are active initially. In superstep 1, \( s \) sends its tag 10 to all \( v \in \Gamma_{out}(s) \), and \( t \) sends its tag 01 to all \( v \in \Gamma_{in}(t) \). They then vote to halt. In superstep \( i > 1 \), a vertex \( v \) computes the bitwise-OR of all messages it receives, which results in \( tag^* \). If \( tag^* = 11 \), or if the bitwise-OR of \( tag^* \) and \( tag(v) \) is 11, \( v \) sets \( tag(v) = 11 \) and calls terminate() to end the program since the two BFSs now meet at \( v \); otherwise, if \( tag(v) = 00 \), \( v \) sets \( tag(v) = tag^* \) and, either sends \( tag^* \) to all \( u \in \Gamma_{out}(v) \) if \( tag^* = 10 \), or sends \( tag^* \) to all \( u \in \Gamma_{in}(v) \) if \( tag^* = 01 \). Finally, \( v \) votes to halt.

Note that if we set \( t \) to be a non-active vertex \( \Gamma_{in}(\ldots,1) \), the algorithm becomes BFS from \( s \). We now analyze the cost of doing BFS. Let \( V_i \) be the set of vertices that are a hop away from \( s \). We can prove by induction that, in superstep \( i \), only those vertices in \( V_i \), both receive messages (from the vertices in \( V_{i-1} \)) and send messages to their out-neighbors. If a vertex in \( V_j \) (\( j < i \)) receives a message, it simply votes to halt without sending messages; while all vertices in \( V_j \) (\( j > i \)) remain inactive as they are not reached yet. Thus, the total workload is \( O(|E| + |V|) \).

Block-centric algorithm. This algorithm operates in VB-mode. In each superstep, V-compute() is first called where each vertex \( v \) receives messages and updates \( tag(v) \) as in the vertex-centric algorithm. If \( tag(v) \) is updated, \( v \) remains active; otherwise, \( v \) votes to halt. Then, B-compute() is called, where each block \( B \) collects all its active vertices with tag 10 (01) into a queue \( Q_s(Q_o) \). If an active vertex with tag 11 is found, \( B \) calls terminate(). Otherwise, \( B \) performs a forward BFS using \( Q_s \) as follows. A vertex \( v \) is dequeued from \( Q_s \) each time, and the out-neighbors \( u \in \Gamma_{out}(v) \) are updated as follows. For each out-neighbor \( u \in V(B) \), if \( tag(u) = 00 \), \( tag(u) \) is set to 10 and \( u \) is enqueued; if \( tag(u) = 01 \) or 11, \( tag(u) \) is set to 11 and terminate() is called. For each out-neighbor
6.3 PageRank

Given a directed graph $G = (V, E)$, the problem is to compute the PageRank of each vertex $v \in V$. Pregel’s PageRank algorithm [11] works as follows. In superstep 1, each vertex $v$ initializes $pr(v) = 1/|V|$ and distributes $pr(v)$ to the out-neighbors by sending each one $pr(v)/\Gamma_{out}(v)$. In superstep $i$ ($i > 1$), each vertex $v$ sums up the received PageRank values, denoted by $sum$, and computes $pr(v) = 0.15/|V| + 0.85 \times sum$. It then distributes $pr(v)/\Gamma_{out}(v)$ to each out-neighbor. A combiner is used, which aggregates the messages to be sent to the same destination vertex into a single message that equals their sum.

**PageRank loss.** Conceptually, the total amount of PageRank values remain to be 1, with 15% held evenly by the vertices, and 85% redistributed among the vertices by propagating along the edges. However, if there exists a sink page $v$ (i.e., $\Gamma_{out}(v) = \emptyset$), $pr(v)$ is not distributed to any other vertex in the next superstep and the value simply gets lost. Therefore, in [11]'s PageRank algorithm, the total amount of PageRank values decreases as the number of supersteps increases.

Let $V_0 = \{ v \in V : \Gamma_{out}(v) = \emptyset \}$ be the set of sink vertices, i.e., vertices with out-degree 0. We ran [11]'s PageRank algorithm on two web graphs (listed in Figure 7) and computed the PageRank loss: (1) WebUK which has 9.08% of the vertices being sink vertices, and (2) WebBase which has 23.41% of the vertices being sink vertices. We found that the algorithm converges in 88 supersteps with 17% PageRank loss on WebUK, and in 79 supersteps with 34% PageRank loss on WebBase. As we shall see shortly, such a large PageRank loss reveals a problem that must be addressed.

**Vertex-centric algorithm.** A common solution to the PageRank loss problem is to make each sink vertex $v \in V_0$ link to all the vertices in the graph, i.e., to distribute $pr(v)/|V|$ to each vertex. Intuitively, this models the behavior of a random surfer: if the surfer arrives at a sink page, it picks another URL at random and continues surfing again. Since $|V|$ is usually large, $pr(v)/|V|$ is small and the impact of $v$ to the PageRank of other vertices is negligible.

Compared with the standard PageRank definition above, Pregel’s PageRank algorithm [11] changes the relative importance order of the vertices. To illustrate, we compute PageRank on the four web graphs from the SNAP database\(^1\) using the algorithm in [11] and the standard PageRank definition, and compare the ordered vertex lists obtained. The results in Figure 5 show that the two vertex lists have a large Kendall tau distance. For example, the graph Google has over 2 million vertex pairs whose order of PageRank magnitude is reversed from the standard definition.

Obviously, materializing $\Gamma_{out}(v) = V$ for each $v \in V_0$ is unacceptable both in space and in communication cost. We propose an aggregator-based solution. In `compute()`, if $v$’s out-degree is 0, $v$ provides $pr(v)$ to an aggregator that computes $agg = \sum_{v \in V_0} pr(v)$.

\(^1\)http://www.google.com/patents/US20080075014
\(^2\)http://snap.stanford.edu/data/
Ties are broken arbitrarily. The set \( V C(s_i) \) is called the Voronoi cell of \( s_i \), and the Voronoi cells of all sources form the GVD of \( G \).

Figure 6(a) illustrates the concept of GVD, where source vertices are marked with solid circles. Consider vertex \( v \) in Figure 6(a), it is at least 2, 3 and 5 hops from the red, green and blue sources. Since the red source is closer to \( v \), \( v \) is assigned to the red Voronoi cell of the red source. All the vertices in Figure 6(a) are partitioned into three Voronoi cells, where the vertices with the same color belong to the same Voronoi cell.

The GVD computation can be easily implemented in the vertex-centric computing model, by performing multi-source BFS. Specifically, in superstep 1, each source \( s \) sets \( block(s) = s \) and broadcasts it to the neighbors; for each non-source vertex \( v \), \( block(v) \) is unassigned. Finally, the vertex votes to halt. In superstep \( i \) (\( i > 1 \)), if \( block(v) \) is unassigned, \( v \) sets \( block(v) \) to an arbitrary source received, and broadcasts \( block(v) \) to its neighbors before voting to halt. Otherwise, \( v \) votes to halt directly. When the process converges, we have \( block(v) = s_i \) for each \( v \) in \( V C(s_i) \).

The multi-source BFS has linear workload since each vertex only broadcasts a message to its neighbors when \( block(v) \) is assigned, and thus the total messages exchanged by all vertices is bounded by \( O(|E|) \). However, we may obtain some huge Voronoi cells (or blocks), which are undesirable for load balancing. We remark that block sizes can be aggregated at the master in a similar manner as when we compute the block-to-worker assignment in Section 4.

Our GVD partitioner works as follows, where we use a parameter \( b_{\text{max}} \) to limit the maximum block size. Initially, each vertex \( v \) samples itself as a source with probability \( p_{\text{samp}} \). Then, multi-source BFS is performed to partition the vertices into blocks. If the size of a block is larger than \( b_{\text{max}} \), we set \( block(v) \) unassigned for any vertex \( v \) in the block (and reactive \( v \)). We then perform another round of source sampling and multi-source BFS on those vertices \( v \) with \( block(v) \) unassigned, using a higher sampling probability. Here, we increase \( p_{\text{samp}} \) by a factor of \( f \) (\( f \geq 1 \)) after each round in order to decrease the chance of obtaining an oversized block. This process is repeated until the stop condition is met.

We check two stop conditions, and the process described above stops as long as one condition is met: (1) let \( A(i) \) be the number of active vertices at the beginning of the \( i \)-th round of multi-source BFS, then the process stops if \( A(i)/A(i-1) > \gamma \). Here, \( \gamma \leq 1 \) is a parameter determining when multi-source BFS is no longer effective in assigning blocks; or (2) the process stops if \( p_{\text{samp}} > p_{\text{max}} \), where \( p_{\text{max}} \) is the maximum allowed sampling rate (recall that \( p_{\text{samp}} = \frac{1}{\gamma} \cdot \text{block after each round} \).

Moreover, to prevent multi-source BFS from running too many supersteps, which may happen if graph diameter \( \delta \) is large and the sampling rate \( p_{\text{samp}} \) is small, we include a user-specified parameter \( \delta_{\text{max}} \) to bound the maximum number of supersteps, i.e., each vertex votes to halt in superstep \( \delta_{\text{max}} \) during multi-source BFS.

When the above process terminates, there may still be some vertices not assigned to any block. This could happen since multi-source BFS works well on large CCs, as a larger CC tends to contain more sampled sources, but the sampling is ineffective for handling small CCs. For example, consider the extreme case where the graph is composed of isolated vertices. Since each round of multi-source BFS assigns block ID to only around \( p_{\text{samp}} |V| \) vertices, it takes around \( \frac{1}{p_{\text{samp}}} \) (which is 1000 if \( p_{\text{samp}} = 0.1\% \)) rounds to assign block ID to all vertices, which is inefficient.

Our solution to assigning blocks for small CCs is by the HashMin algorithm, which marks each small CC as a block using only a small number of supersteps. Specifically, after the rounds of multi-source BFS terminate, if there still exists any unassigned vertex, we run HashMin on the subgraph induced by these unassigned vertices. We call this step as subgraph HashMin.

There are six parameters, \( (p_{\text{samp}}, \delta_{\text{max}}, b_{\text{max}}, f, \gamma, p_{\text{max}}) \), that need to be specified for the partitioner. We show that these parameters are intuitively easy to set. In particular, we found that the following setting of the parameters work well for most large real-world graphs. The sampling rate \( p_{\text{samp}} \) decides the number of blocks, and usually a small value as 0.1% is a good choice. Note that \( p_{\text{samp}} \) cannot be too small in order not to create very large blocks. As for the stopping parameters, \( \gamma \) is usually set as 90%, and \( p_{\text{max}} \) as 10% with \( f = 2 \), so that there will not be too many rounds of multi-source BFS. The bound on the number of supersteps, \( \delta_{\text{max}} \), is set to be a tolerable number such as 50, but for small-diameter graphs (e.g., social networks) we can set a smaller \( \delta_{\text{max}} \) such as 10 since the number of supersteps needed for such graphs is small. We set \( b_{\text{max}} \) to be 100 times that of the expected block size (e.g., \( b_{\text{max}} = 100,000 \) when \( p_{\text{samp}} = 0.1\% \)) for most graphs, except that for spatial networks the block size is limited by \( \delta_{\text{max}} \) already and hence we just set it to \( \infty \). We will further demonstrate in our experiments that the above settings work effectively for different types of real-world graphs.

### 7.2 2D Partitioner

In many spatial networks, vertices are associated with \((x, y)\)-coordinates. Bioler provides a 2D partitioner to partition such graphs. A 2D partitioner associates each vertex \( v \) with an additional field \((x, y)\), and it consists of two jobs.

The first job is vertex-centric and works as follows: (1) each worker samples a subset of its vertices with probability \( p_{\text{samp}} \) and sends the sample to the master; (2) the master first partitions the sampled vertices into \( n_x \) slots by the \( x \)-coordinates, and then each slot is further partitioned into \( n_y \) slots by the \( y \)-coordinates. Each resulting slot is a super-block.Figure 6(b) shows a 2D partitioning with \( n_x = n_y = 3 \). This partitioning is broadcast to the workers, and each worker assigns each of its vertices to a super-block according to the vertex coordinates. Finally, vertices are exchanged according to the super-block-to-worker assignment computed by master, with \( \tilde{\Gamma}(v) \) constructed for each vertex \( v \) as described in Section 4.

Since a super-block may not be connected, we perform a second block-centric job, where workers run BFS over their super-blocks to break them into connected blocks. Each worker marks the blocks it obtains with IDs \( 0, 1, \ldots \). To get the global block ID, each worker \( w_j \) sends the number of blocks it has, denoted by \( |\mathcal{B}_j| \), to the master which computes for each worker \( w_j \) a prefix sum \( \text{sum}_j = \sum_{i<j} |\mathcal{B}_i| \), and sends \( \text{sum}_j \) to \( w_j \). Each worker \( w_j \) then adds \( \text{sum}_j \) to the block IDs of its blocks, and hence each block obtains a unique block ID. Finally, \( \tilde{\Gamma}(v) \) are updated for each vertex \( v \).

The 2D partitioner has parameters \( (p_{\text{samp}}, n_x, n_y) \). The setting of \( p_{\text{samp}} \) is similar to the GVD partitioner. To set \( n_x \) and \( n_y \), we use \( \delta(G) \approx O(\sqrt{\max\{n_x, n_y\}}) \) as a guideline, since the diameter of \( G \) is critical to the performance of block-centric algorithms.

### 8. EXPERIMENTS

We compare the performance of Bioler and with Giraph 1.0.0, Giraph++\(^1\) and GraphLab 2.2 (which includes the features of PowerGraph [4]). We ran our experiments on a cluster of 16 machines, each with 24 processors (two Intel Xeon E5-2620 CPU) and 48GB RAM. One machine is used as the master that runs only one worker, while the other 15 machines act as slaves running multiple workers. The connectivity between any pair of nodes in the cluster is 1Gbps.

---

1 [https://issues.apache.org/jira/browse/GIRAPH-818](https://issues.apache.org/jira/browse/GIRAPH-818)
For the experiments of Giraph, we use the multi-threading feature added by Facebook, and thus each user refers to a computing thread. However, Giraph++ is built on top of an earlier Giraph version by Yahoo!, which does not support multi-threading. We therefore run multiple workers (mapper tasks) per machine. We make all Giraph++ codes used in our experiments public.

We used seven large real-world datasets, which are from four different domains as shown in Figure 7: (1) web graphs: WebUK and WebBase; (2) social networks: Friendster and LiveJournal; (3) RDF graph: BTC (a graph converted from the Billion Triple Challenge 2009 RDF dataset [2]); (4) road networks: USA and Euro.

Among them, WebUK, LiveJournal and BTC have skewed degree distribution; WebUK, Friendster and LiveJournal have average degree relatively higher than other large real-world graphs; USA and Euro, as well as WebUK, have a large diameter.

For a graph of certain size, we need a certain amount of computing resources (i.e., workers) to achieve good performance. However, the performance does not further improve if we increase the number of workers per slave machine beyond that amount, since the increased overhead of inter-machine communication outweighs the increased computing power. In the experiments, we run 10 workers for WebUK and WebBase, 8 for Friendster, 2 for LiveJournal, 4 for BTC, USA and Euro, which exhibit good performance.

8.1 Blogel Implementation

We make Blogel open-source. All the system source codes, as well as the source codes of the applications discussed in this paper, can be found in http://www.cse.cuhk.edu.hk/blogel.

Blogel is implemented in C++ as a group of header files, and users only need to include the necessary base classes and implement the application logic in their subclasses. Blogel communicates with HDFS through libhdfs, a JNI based C API for HDFS. Each worker is simply an MPI process and communications are implemented using MPI communication primitives. While one may deploy Blogel with any Hadoop and MPI version, we use Hadoop 1.2.1 and MPICH 3.0.4 in our experiments. All programs are compiled using GCC 4.4.7 with -O2 option enabled.

Blogel makes the master a worker, and fault recovery can be implemented by a script as follows. The script loops a Blogel job, which runs for at most 150 supersteps before dumping the intermediate results to HDFS. Meanwhile, the script also monitors the cluster condition. If a machine is down, the script kills the current job and restarts another job loading the latest intermediate results from HDFS. Fault tolerance is achieved by the data replication in HDFS.

8.2 Performance of Partitions

We first report the performance of Blogel’s partitioners.

### Table 7: Datasets

<table>
<thead>
<tr>
<th>Data</th>
<th>Type</th>
<th>V</th>
<th>E</th>
<th>AVG Deg</th>
<th>Max Deg</th>
</tr>
</thead>
<tbody>
<tr>
<td>WebUK</td>
<td>directed</td>
<td>133,633,040</td>
<td>5,597,076,822</td>
<td>41.21</td>
<td>22,429</td>
</tr>
<tr>
<td>WebBase</td>
<td>directed</td>
<td>118,142,355</td>
<td>11,819,937,190</td>
<td>8.63</td>
<td>3,541</td>
</tr>
<tr>
<td>Friendster</td>
<td>undirected</td>
<td>65,608,366</td>
<td>3,612,134,270</td>
<td>55.06</td>
<td>5,214</td>
</tr>
<tr>
<td>RDB</td>
<td>directed</td>
<td>10,690,276</td>
<td>224,614,770</td>
<td>21.01</td>
<td>1,053,676</td>
</tr>
<tr>
<td>BTC</td>
<td>undirected</td>
<td>164,732,473</td>
<td>772,222,094</td>
<td>4.69</td>
<td>6,837,619</td>
</tr>
<tr>
<td>USA Road</td>
<td>undirected</td>
<td>23,947,347</td>
<td>38,333,344</td>
<td>2.44</td>
<td>9</td>
</tr>
<tr>
<td>Euro Road</td>
<td>undirected</td>
<td>18,029,721</td>
<td>44,826,904</td>
<td>2.49</td>
<td>12</td>
</tr>
</tbody>
</table>

### Figure 8: Performance of GVD partitioners

Figure 8 shows the performance of the GVD partitioners, along with the parameters we used. The web graphs can be partitioned simply based on URLs, but we also apply GVD partitioners to them for the purpose of comparison. The partitioning parameters are set according to the heuristics given in Section 7, but if there exists a giant block in the result which is usually obtained in the phase when Hash-Min is run, we check why the multi-source BFS phase terminates. If it terminates because the sampling rate increases beyond $p_{\text{max}}$, we decrease $f$ and increase $p_{\text{max}}$ and $\gamma$ to allow more rounds of multi-source BFS to be run in order to break the giant block into smaller ones. We also increase $b_{\text{max}}$ to relax the maximum block size constraint during multi-source BFS, which may generate some relatively larger blocks (still bounded by $b_{\text{max}}$) but reduce the chance of producing a giant block (not bounded by $b_{\text{max}}$). We slightly adjusted the parameters for WebUK, WebBase and BTC in this way, while all other datasets work well with the default setting described in Section 7.

Recall from Section 4 that besides block-to-worker assignment, partitioners also compute vertex-to-block assignment (denoted by “Block Assign.” in Figure 8), construct $\Gamma(v)$ (denoted by “Triplet ID Neighbors”), and exchange vertices according to the block-to-worker assignment (denoted by “Vertex Exchange”). We report the computing time for “Block Assign.”, “Triplet ID Neighbors”, and “Vertex Exchange” in Figure 8. We also report the data loading/dumping time and the total computation time of the GVD partitioners in the gray columns. As for vertex-to-block assignment, (1) for multi-source BFS, we report the number of rounds, the total number of supersteps taken by all these rounds, and the average time of a superstep; (2) for running Hash-Min on the subgraphs, we report the number of supersteps and the average time of a superstep.

As Figure 8 shows, the partitioning is very efficient as the computing time is comparable to graph loading/dumping time. In fact, within the overall computing time, a significant amount of time is spent on constructing $\Gamma(v)$ and vertex exchange, and the vertex-to-block assignment computation is highly efficient. Figure 9 shows the number of blocks and vertices assigned to each worker, for example, WebUK is partitioned over workers 0 to 150 and each worker contains $x$ blocks and $y$ vertices, where $16,781 \leq x \leq 16,791$ and $884,987 \leq y \leq 884,988$. Thus, we can see that the GVD partitioners achieve very balanced block-to-worker assignment, and it also shows that the greedy algorithm described in Section 4 is effective. The workload is relatively less balanced only for BTC, where we have 13 larger blocks distributed over workers 0 to 12. This is mainly caused by the few vertices in BTC with very high degree (the max degree is 1,652). We remark that probably no general-purpose partitioning algorithm is effective on BTC, if there exists a balanced partitioning for BTC at all.

### Performance of 2D partitioners

Since the vertices in USA and Euro have 2D coordinates, we also run 2D partitioners on them, with $(f_{\text{amp}}, n_{x}, n_{y}) = (1\% , 20, 20)$. Figure 10(a) shows the performance of 2D partitioning, which reports the number of super-blocks, blocks and vertices in each worker. The number of blocks/vertices per worker obtained by 2D partitioning is not as even as that by GVD partitioning. However, the shape of the super-blocks are very regular, resulting in fewer number of edges crossing super-blocks and a smaller diameter for the block-level graph, and thus block-centric algorithms can run faster.

Figures 10(b) and 10(c) further show that 2D partitioning is also more efficient than GVD partitioning. For example, for USA, job 1 of 2D partitioning takes 26.92 seconds (58% spent on loading and dumping) and job 2 takes 13.29 seconds (93.5% spent on loading and dumping), which is still much shorter than the time taken by the GVD partitioner (9.06 + 38.56 + 7.77 = 55.39 seconds).
Comparison with existing partitioning methods. One of the widely used graph partitioning algorithms is METIS [7] (e.g., GRACE [20] uses METIS to partition the input graph). However, METIS ran out of memory on those large graphs in Figure 7 on our platform. To solve the scalability problem of METIS, Giraph++ [18] proposed a graph coarsening method to reduce the size of the input graph so that METIS can run on the smaller graph. Here, a vertex in the coarsened graph corresponds to a set of connected vertices in the original graph. The partitioning algorithm of Giraph++ consists of 4 phases: (1) graph coarsening, (2) graph partitioning (using METIS), (3) graph uncoarsening, which projects the block information back to the original graph, and (4) ID recoding, which relabels the vertex IDs so that the worker of a vertex v can be obtained by hashing v’s new ID. Note that ID recoding is not required in Blogel since the worker ID of each vertex is stored in its triplet ID triple(v). This approach retains the original vertex IDs so that Blogel’s graph computing results require no ID re-projection. Blogel’s graph partitioning is also more user-friendly, since it requires only one partitioner job, while Giraph++’s graph partitioning consists of a sequence of over 10 Giraph/MapReduce/METIS jobs.

Figure 11 shows the partitioning performance of Giraph++, where we ran as many workers per machine as possible (without running out of memory). We did not obtain result for WebUK and Friendster, since graph coarsening ran out of memory even when each slave machine runs only one worker. Figure 11 shows that the partitioning time of Giraph++ is much longer than that of our GVD partitioner. For example, while our GVD partitioner partitions WebBase in 334.94 seconds (see the breakdown time in Figure 8), Giraph++ uses 5450 seconds. In general, our GVD partitioner is tens of times faster than Giraph++’s METIS partitioning algorithm.

Recently, Stanton and Kliot [17] proposed a group of algorithms to partition large graphs, and the best one is a semi-streaming algorithm called Linear (Weighted) Deterministic Greedy (LDG). We also ran LDG and the results are presented in Figure 12. We can see that LDG is many times slower than our GVD partitioner (reported in Figure 8), though LDG is much faster than Giraph++’s new METIS partitioning algorithm.

8.3 Partitioner Scalability
We now study the scalability of our GVD partitioner. We first test the partitioning scalability using two real graphs: BTC with skewed degree distribution, and USA with a large graph diameter. We partition both graphs using varying number of slave machines (i.e., 6, 9, 12 and 15), and study how the partitioning performance scales with the amount of computing resources. We report the results in Figure 13. For the larger graph BTC, we need a certain amount of computing resources to achieve good performance. For example, when the number of slave machines increases from 6 to 9, the partitioning time of BTC improves by 25.7%, from 189.2 seconds to 140.52 seconds. However, the performance does not further improve if we increase the number of machines beyond 12. This is because the increased overhead of inter-machine communication outweighs the increased computing power. For the relatively smaller graph USA, the performance does not change much with varying number of slave machines, since the computing power is sufficient even with only 6 slaves.

To test the scalability of our GVD partitioner as the graph size increases, we generate random graphs using PreZER algorithm [12]. We set the average degree as 20 and vary |V| to be 25M, 50M, 75M and 100M. Figure 14 shows the scalability results, where all the 16 machines in our cluster are used. The partitioning time increases almost linearly with |V|, which verifies that our GVD partitioner scales well with graph size. Moreover, even for a graph with |V| = 100M (i.e., |E| ≈ 2|V|), the partitioning is done in
our block-centric system, \( B-GVD \) or \( B-2D \), is one to two orders of magnitude faster than the vertex-centric systems, \( V-Centric \), Giraph and GraphLab, for processing the large-diameter graphs. But the improvement is limited for the small-diameter graph, since \( LiveJournal \) is not very large and the vertex-centric systems are already very fast on a small-diameter graph of medium size. Compared with Giraph++, \( B-GVD \) is significantly more efficient for processing \( LiveJournal \) and \( B-2D \) is much faster for processing \( USA \), while Giraph++ failed to run on \( WebUK \).

For PageRank, we use the two web graphs \( WebUK \) and \( WebBase \). We use both the URL-based partitioner and the GVD partitioner for graph partitioning. Figure 16(d) shows the number of blocks/vertices per worker using URL partitioning, where we see that URL partitioning achieves more balanced workload and less number of blocks than GVD partitioning (cf. Figure 9). This shows that background knowledge about the graph data can usually offer a higher quality partitioning than a general method.

We report the time of computing local PageRank and BlockRank by Blogel’s B-mode in Figure 16(a). We present the average time of a superstep and the total number of supersteps of computing PageRank using different systems in Figure 16(b). We also run Blogel’s block-centric V-mode with the input graph partitioned by LDG partitioning [17], denoted by \( B-LDG \) in Figure 16(b). We remark that LDG cannot be used in Blogel’s VB-mode and B-mode, since a partition obtained by LDG is not guaranteed to be connected.

The results show that though running V-mode, block-centric computing is still significantly faster than vertex-centric computing (i.e., \( V-Centric \), Giraph and GraphLab). Note that \( B-GVD \) and \( B-URL \) are also running block-centric V-mode in Figure 16(b). Thus, the result also reveals that our GVD partitioner leads to more efficient distributed computing than LDG. \( B-URL \) is comparable with \( B-LDG \) on \( WebUK \), but is significantly faster than \( B-LDG \) on \( WebBase \). The superior performance of \( B-GVD \) and \( B-URL \) is mainly because the GVD and URL partitioner achieve greater reduction in the number of cross-worker edges than LDG, which results in less number of messages exchanged through the network.

We also notice that the number of supersteps of the block-centric algorithm is more than that of the vertex-centric algorithm, which is mainly due to the fact that the PageRank initialization formula of \( [6] \), i.e., \( pr(v) = \frac{lpr(v) - br(block(v))}{lpr(v)} \), is not effective. We may improve the algorithm by specifying another initialization formula, but this is not the focus of this paper.

Another kind of PageRank algorithm is adopted in Giraph++’s paper [18], which is based on the accumulative iterative update approach of [22]. To make a fair comparison with Giraph++, we also developed a Blogel vertex-centric counterpart, and a block-centric counterpart that runs in VB-mode. As Figure 16(c) shows, Blogel’s block-centric computing (i.e., \( B-GVD \) or \( B-URL \)) is significantly faster than its vertex-centric counterpart (i.e., \( V-Centric \)) when accumulative iterative update is applied. We can only compare with Giraph++ on \( WebBase \) since Giraph++ failed to partition \( WebUK \). Figure 16(c) shows that while Giraph++ is twice faster than \( V-Centric \), it is still much slower than \( B-GVD \) and \( B-URL \).

### 9. CONCLUSIONS

We presented a block-centric framework, called Blogel, and showed that Blogel is significantly faster than existing distributed graph computing systems \([1, 10, 4, 18]\), for processing large graphs with adverse graph characteristics such as skewed degree distribution, high average degree, and large diameter. We also showed that Blogel’s partitioners generate high-quality blocks and are much faster than the state-of-the-art graph partitioning methods \([17, 18]\).

### Figure 14: GVD Partitioner Scalability on Random Graphs

| | Machine # |
|---|---|---|---|---|
| | 6 | 9 | 12 | 15 |
| Load | 11.44 s | 29.05 s | 42.10 s | 74.31 s |
| Compute | 45.32 s | 88.76 s | 127.32 s | 164.13 s |
| Dump | 22.26 s | 41.64 s | 58.86 s | 76.84 s |
| Total | 79.02 s | 159.45 s | 228.29 s | 315.28 s |

### Figure 13: GVD Partitioner Scalability on Real Graphs

<table>
<thead>
<tr>
<th></th>
<th>BTC</th>
<th>USA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>66.35 s</td>
<td>68.50 s</td>
</tr>
<tr>
<td>Compute</td>
<td>189.20 s</td>
<td>140.52 s</td>
</tr>
<tr>
<td>Dump</td>
<td>59.13 s</td>
<td>52.96 s</td>
</tr>
<tr>
<td>Total</td>
<td>208.69 s</td>
<td>231.98 s</td>
</tr>
</tbody>
</table>

| | | | | |
|---|---|---|---|
| | 25M | 50M | 75M | 100M |
| Load | 11.44 s | 29.05 s | 42.10 s | 74.31 s |
| Compute | 45.32 s | 88.76 s | 127.32 s | 164.13 s |
| Dump | 22.26 s | 41.64 s | 58.86 s | 76.84 s |
| Total | 79.02 s | 159.45 s | 228.29 s | 315.28 s |
For future work, we plan to define a class of algorithms similar to PPA [21] for the block-centric computing model.

Acknowledgments.
We thank the reviewers for giving us many constructive comments, with which we have significantly improved our paper. This work was partially done when the first author was at HKUST. This research is supported in part by SHIAE Grant No. 8115048 and HKUST Grant No. FSGRF14EG31.

10. REFERENCES