GPU-Accelerated Graph Clustering via Parallel Label Propagation

Yusuke Kozawa*
Artificial Intelligence Research Center, AIST, Japan
yusuke.kozawa@aist.go.jp

Toshiyuki Amagasa
Center for Computational Sciences, University of Tsukuba, Japan
amagasa@cs.tsukuba.ac.jp

Hiroyuki Kitagawa
Center for Computational Sciences, University of Tsukuba, Japan
kitagawa@cs.tsukuba.ac.jp

ABSTRACT

Graph clustering has recently attracted much attention as a technique to extract community structures from various kinds of graph data. Since available graph data becomes increasingly large, the acceleration of graph clustering is an important issue for handling large-scale graphs. To this end, this paper proposes a fast graph clustering method using GPUs. The proposed method is based on parallelization of label propagation, one of the fastest graph clustering algorithms. Our method has the following three characteristics: (1) *efficient parallelization*: the algorithm of label propagation is transformed into a sequence of data-parallel primitives; (2) *load balance*: the method takes into account load balancing by adopting the primitives that make the load among threads and blocks well balanced; and (3) *out-of-core processing*: we also develop algorithms to efficiently deal with large-scale datasets that do not fit into GPU memory. Moreover, this GPU out-of-core algorithm is extended to simultaneously exploit both CPUs and GPUs for further performance gain. Extensive experiments with real-world and synthetic datasets show that our proposed method outperforms an existing parallel CPU implementation by a factor of up to 14.3 without sacrificing accuracy.

1 INTRODUCTION

Graph clustering, also known as *community detection*, is a technique to extract community structures (or *clusters*) from graph-structured data, such that vertices in one cluster are densely connected and vertices in different clusters are sparsely connected [12]. Graph clustering has been successfully applied for various kinds of graph data ranging from the Web and online social networks to biological data [21]. However, recent graph data becomes increasingly large, and it is difficult to handle such large-scale data by existing methods in realistic time. Thus it is highly important to accelerate the performance of graph clustering.

*Part of the work was done while the author was a student at the University of Tsukuba.

CIKM '17, November 6–10, 2017, Singapore.
© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-4918-5/17/11…$15.00
https://doi.org/10.1145/3132847.3132960

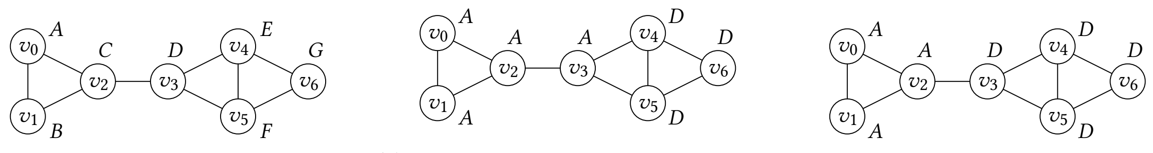
Among existing algorithms, *label propagation*[1] is known as one of the fastest algorithms [22]. It has the following three attractive features: (1) the computational complexity is linear to the number of edges, which is faster than many other algorithms; (2) it is reported that the clustering accuracy of label propagation is also better than other state-of-the-art algorithms [26]; and (3) the algorithm is suitable for parallel processing [16], because it makes clusters by only using vertex-local (i.e., adjacent vertices) information. Thus, parallel label propagation is considered as a promising solution for accelerating graph clustering. In particular, GPUs are the ideal platforms to this end, as they provide tremendous computing performance with massive parallelism and very high memory bandwidth.

Label propagation using GPUs has already been proposed by Soman and Narang [23]; however, their method has the limitation that it does not well take into account *load balancing*. In general, there exist several challenges to harness the power of GPUs. Among them, load balancing is a particularly important issue for graph clustering, given the *scale-free* property of real-world graphs and the massive parallelism of GPUs. A scale-free graph is a graph whose degree distribution follows a power law [2]. This means that most vertices have low degrees, while few vertices have extremely high degrees. Therefore, if the algorithm is naïvely parallelized (e.g., by assigning the work of one vertex to one worker), the implementation suffers from severe load imbalance.

In this paper, we present a novel GPU-based method to accelerate graph clustering by parallelizing label propagation. To realize efficient GPU-accelerated label propagation, three challenges need to be addressed: (1) fast label counting on massively parallel architectures; (2) load balancing; and (3) efficient handling of large-scale data that does not fit into GPU memory.

The first and second challenges are addressed by combining multiple *data-parallel primitives* [13]. The label propagation algorithm requires counting labels (or cluster IDs) of neighbors of each vertex. To this end, associative arrays such as hash tables are commonly employed [24]. However, their efficient implementations for GPUs are generally difficult because of the massive parallelism. Instead, we transform the counting procedure into a series of data-parallel primitives, thereby leveraging the computational power of GPUs. Although this transformation comes at the expense of accuracy, this issue can be remedied by introducing the concept of *semi-synchronous* label propagation [8, 10]. Load balancing is also achieved as well by using the primitives that make the load among processors almost completely balanced, even if degree distributions are heavily skewed.

[1]While the term "label propagation" has more general meanings, in this paper it specifically refers to the label propagation algorithm for graph clustering.



(a) The initial state. **(b) Labels have been updated in the order (v_2, v_4, v_6, v_1, v_3, v_0, v_5).** **(c) The steady state.**

Figure 1: Example of label propagation. The letters near vertices denote the labels.

The third challenge should be also overcome, because large-scale real-world graph data tends to contain billions of edges, while GPUs have memory of only a dozen gigabytes. To this end, we partition graph data on the CPU and overlap the computation and data transfers to the GPU, thereby hiding the data-transfer latency, which commonly becomes the bottleneck in large-scale GPU applications. Besides, this GPU out-of-core method is extended to exploit both CPUs and GPUs simultaneously.

In summary, this paper makes the following contributions:

- We express the algorithm of label propagation as a series of operations that are readily parallelizable on many-core processors like GPUs.
- We balance the load among processors by exploiting *skew-resistant* operators such as segmented sort and segmented reduce.
- We develop GPU out-of-core implementations that are capable of dealing with larger data than the size of GPU memory, without sacrificing performance or even faster than in-core implementations. The out-of-core implementation is further extended to a CPU–GPU hybrid version.

We conducted extensive experiments for evaluating the performance of our GPU-based methods by using real-world and synthetic datasets. The results reveal that our proposed methods not only outperform an existing parallel CPU implementation by a factor of up to 14.3, but also achieve high accuracy to the similar extent with serial label propagation.

The rest of this paper is organized as follows. Section 2 describes preliminary knowledge. Section 3 provides our proposed methods, and Section 4 experimentally evaluates them with comparison to other implementations. Section 5 briefly reviews related work of serial and parallel graph clustering. Finally, Section 6 concludes this paper.

Algorithm 1: Label propagation

```
   Input: A graph $G = (V, E)$
   Output: A set $C$ of clusters
1  Initialize label assignment $L$  ▹ $L_i$ is the label of a vertex $v_i$
2  repeat
3      $\pi \leftarrow$ a random permutation of $(0, 1, \ldots, n-1)$
4      for $i \in \pi$ do
5          $L_i \leftarrow \arg\max_{l \in L} \left|\{j \mid (v_i, v_j) \in E \wedge L_j = l\}\right|$
6      end
7  until A termination condition is met
8  $C \leftarrow$ extract clusters
9  return $C$
```

2 PRELIMINARIES

This section presents basic concepts and an existing algorithm: graph clustering (Section 2.1), label propagation (Section 2.2), and GPU computing (Section 2.3).

2.1 Graph clustering

Let $G = (V, E)$ be a graph, where V is the set of vertices and $E \subset V \times V$ is the set of edges. We denote the numbers of vertices and edges as n and m, respectively. For the sake of simplicity, this work assumes that graphs are undirected and unweighted, but the proposed method can be easily extended to handle directed and weighted graphs. The problem tackled in this paper, *graph clustering*, is informally stated as follows: given a graph $G = (V, E)$, find a partition of V, $C = \{C_1, C_2, \ldots, C_l\}$, where C_i is called *cluster*; vertices in the same cluster should be densely connected, while vertices between different clusters should be sparsely connected.

In this paper, we assume that the input graph data is available on the CPU main memory before clustering, and our goal is to speed up generating a clustering result on the main memory.

2.2 Label propagation

Label propagation, proposed by Raghavan et al. [22], is one of the fastest graph-clustering algorithms. It clusters a graph by the assumption that connected vertices tend to be members of the same cluster. Algorithm 1 shows a pseudo-code of label propagation. First, vertices are initialized by unique cluster labels (Line 1). Then the labels of vertices are to be updated in random order (Lines 3–6). A label of a vertex v_i is updated to the label that most frequently appears among the adjacent vertices of v_i (Line 5). If multiple labels occur the same number of times, the tie is broken randomly. This update iteration continues until some termination condition is met, and vertices with the same label are extracted as elements of the same cluster (Lines 8–9). Several termination conditions can be considered, such that the number of updated labels in one iteration becomes sufficiently small, or the number of iterations exceeds a threshold.

Figure 1 illustrates an example of label propagation. First, vertices are associated with unique labels (Figure 1(a)). The labels are to be updated in random order of vertices. In this example, let the order be $(v_2, v_4, v_6, v_1, v_3, v_0, v_5)$. The first vertex v_2 has three adjacent vertices v_0, v_1, and v_3, whose labels are *A*, *B*, and *D*, respectively. In this case, all of the labels appear the same number of times. Thus the label of v_2 is randomly updated to one of the three labels, and it turns out to be *A*. The label of the next vertices are similarly updated. Figure 1(b) shows the situation that all the labels are updated once.

When all vertices are iterated once again (Figure 1(c)), the labels become fixed (i.e., labels can be no longer updated). Finally, two clusters, $C_A = \{v_0, v_1, v_2\}$ and $C_D = \{v_3, v_4, v_5, v_6\}$, are obtained as the clustering result.

2.3 GPU computing

GPU computing means the use of GPUs for accelerating general-purpose computations rather than graphics tasks, which are the original targets of GPUs [20]. For developing GPU applications, CUDA [19] is the de-facto standard framework; thus we focus on CUDA in this work. The following describes several features of GPU architecture and threading model, as well as several data-parallel primitives.

The CUDA GPU architecture is made up of multiple *streaming multiprocessors (SMs)*, which in turn consist of many simple processing units called *scalar processors (SPs)*. CUDA provides fine-grained parallelism by launching a massive number of lightweight threads. Threads are managed and scheduled in groups of 32 parallel threads called *warps*, and multiple warps form a *thread block* (or a *block* for short). Threads within a block run concurrently on an SM, sharing the resources of the SM. On the other hand, an SM can accommodate multiple blocks simultaneously, maintaining its resources among blocks and scheduling the threads of blocks. Blocks comprise a *grid*, which is generated each time when functions to be executed on GPUs are invoked; such functions are referred to as *kernels*.

GPUs have several kinds of memory. The largest memory on GPUs is *global memory*. For instance, NVIDIA Titan X (Pascal) has the global memory of 12 GB. It can be accessible from all threads of a grid and has a high bandwidth although the access latency is high. SMs also contain local memory, which can be accessed much faster than global memory although the size is small (up to 96 KB, depending on the microarchitecture of GPUs). This memory can be used as *shared memory*, which is shared by threads within a block and can be used to exchange data among the threads. *Registers* are the fastest memory private to each thread with relatively large space (256 KB per SM).

Data-parallel primitives are fundamental operations in data-parallel computing [13]. They can be used as building blocks to efficiently parallelize more complex algorithms, because implementations of primitives are highly tuned and deliver high performance. The rest of this subsection summarizes primitives used in our method: gather, scan, segmented sort, and segmented reduce.

Gather. A *gather* operation carries out indexed reads from an array [13]. It takes two arrays A of n elements and B of m elements and returns the array C of m elements that are computed as $C[i] \leftarrow A[B[i]]$.

Scan. A *scan* [13], also known as *prefix sum*, operation takes an array $[a_0, a_1, \dots, a_{n-1}]$ of n elements and a binary associative and commutative operator $\oplus$ with identity I, and returns the array $\left[I, a_0, a_0 \oplus a_1, \dots, a_0 \oplus a_1 \oplus \cdots \oplus a_{n-2}\right]$.

Segmented sort and segmented reduce. *Segmented sort* and *segmented reduce* are parts of the library *Modern GPU* [4], efficiently working on multiple irregular-length segments within one array in parallel. A segmented sort operation takes a data array $A = \left[a_0, a_1, \dots, a_{n-1}\right]$ and a segment array $S = \left[s_0, s_1, \dots, s_{m-1}, s_m = n\right]$, and sorts each subarray $\left[a_{s_i}, \dots, a_{s_{i+1}-1}\right]$. A segmented reduce operation takes two arrays $A = \left[a_0, a_1, \dots, a_{n-1}\right]$ and $S = \left[s_0, s_1, \dots, s_{m-1}, s_m = n\right]$ and a binary associative and commutative operator $\oplus$. Then it computes the array $\left[r_0, r_1, \dots, r_{m-1}\right]$ where an element r_i is the result of reduce with $\oplus$ over the subarray $\left[a_{s_i}, \dots, a_{s_{i+1}-1}\right]$ of A (i.e., $r_i = a_{s_i} \oplus \cdots \oplus a_{s_{i+1}-1}$).

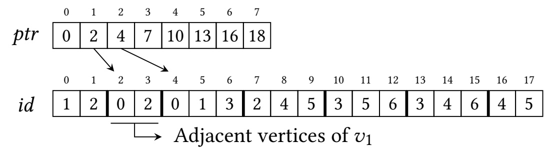


Figure 2: Graph data on the GPU. A graph is represented by two arrays *ptr* and *id*.

Since a simple implementation that distributes work per subarray is very inefficient owing to load imbalance, the implementations of Modern GPU intelligently parallelize the computations so that almost complete load balancing can be achieved.

3 PROPOSED METHOD

This section describes the proposed method that is based on parallelization of label propagation for GPUs. The proposed method has three attractive properties: (1) *Efficient parallelization*: Label propagation requires to accumulate the labels of adjacent vertices (or *adjacent labels* for short) of a vertex v, and computes the majority label for updating the label of v. The proposed method executes this computation efficiently on the GPU by exploiting multiple data-parallel primitives. (2) *Load balance*: If we simply assign a thread block to a vertex, the performance significantly degrades because of load imbalance, given the scale-free property of real-world graphs. Thus, the proposed method parallelizes several operations by utilizing skew-resistant operators, whose performance is not affected by the existence of skewness. (3) *Out-of-core algorithms*: Considering recent large-scale graph data and relatively small GPU memory, it is common that entire graph data cannot fit into the GPU global memory. Hence, we devise a GPU out-of-core algorithm for efficiently handling large-scale data. This out-of-core algorithm can also employ not only GPUs but also CPUs, with slight modifications.

In this section, Section 3.1 introduces the data layout on the GPU. Section 3.2 describes the algorithm of label propagation parallelized for the GPU. Section 3.3 explains GPU out-of-core algorithms to deal with large-scale graph data.

3.1 Data layout

To perform label propagation, the following data needs to be maintained on the GPU: (1) input graph data and (2) vertex labels. Our method represents the input graph data by the *CSR (compressed sparse row)* format, which is a format for sparse matrices [5], because a graph can be regarded as a matrix and it is usually sparse. The advantages of the CSR format, compared to other sparse-matrix formats, are low space cost and its aptitude for data-parallel primitives. Meanwhile, the label information is simply stored as an array of integers where an element is the label of a corresponding vertex. Note that, although the examples of this paper denote the labels as letters for the sake of explanation, labels are stored as integers in the implementation.

Algorithm 2: Parallel label propagation on the GPU

```
   Input: Graph data, ptr and id
   Output: Label assignment L
1  for $i = 0 \to n-1$ in parallel do
2      $L[i] \leftarrow i$                          ▷ Initialize labels
3  end
4  repeat
5      $L' \leftarrow$ gather($L$, $id$)              ▷ Obtain adjacent labels
6      segmented_sort($L'$, $ptr$)
7      for $i = 0 \to m-2$ in parallel do
8          if $L[i] \neq L[i+1]$ then
9              $F[i] \leftarrow 1$                  ▷ $i$ is a boundary
10         else
11             $F[i] \leftarrow 0$
12         end
13     end
14     for $i = 1 \to n$ in parallel do
15         $F[ptr[i]-1] \leftarrow 1$           ▷ $ptr[i]-1$ is also a boundary
16     end
17     $F' \leftarrow$ scan($F$, +, 0)
18     for $i = 0 \to m-1$ in parallel do
19         if $F'[i] \neq F'[i+1]$ then
20             $S[F'[i]] \leftarrow i$              ▷ Boundary index
21         end
22     end
23     for $i = 0 \to n$ in parallel do
24         $Sptr[i] \leftarrow F'[ptr[i]]$
25     end
26     for $i = 1 \to Sptr[n]-1$ in parallel do
27         $W[i] \leftarrow S[i+1] - S[i]$          ▷ Label frequency
28     end
29     $W[0] \leftarrow S[0] + 1$
30     $(W_{\max}, I) \leftarrow$ segmented_reduce($W$, $Sptr$, max)
31     for $i = 0 \to n-1$ in parallel do
32         $L[i] \leftarrow L'[S[I[i]]]$            ▷ Update labels
33     end
34 until A termination condition is met
35 return L
```

Figure 2 illustrates an example of graph data by the CSR format. A graph is represented by two arrays *ptr* and *id*. The array *id* contains the indices of adjacent vertices. The array *ptr* maintains the offsets where the indices of adjacent vertices for a specific vertex start in the array *id*. For example, the information of the adjacent vertices of v_1 in Figure 1 is stored as the elements of the array *id* from $ptr[1] = 2$ (inclusive) to $ptr[2] = 4$ (exclusive). In general, the CSR format represents a sparse matrix with one more array to store the value of each entry. However, since entries of adjacent matrices of unweighted graphs are binary, such an array is omitted in this case.

3.2 Label propagation on the GPU

Algorithm 2 shows a pseudo-code of parallel label propagation on the GPU, which takes graph data as input and computes the label assignment for the vertices. The algorithm first initialize the labels, and proceeds to the main loop. The major computation performed in the loop is to count the occurrences of adjacent labels (Lines 5–29). By using the frequencies, the labels can be updated (Lines 30–33). The following describes the details of these computations.

Counting occurrences. The approach for counting adjacent labels consists of the following four steps: (1) gathering adjacent labels, (2) sorting the adjacent labels per vertex, (3) extracting boundaries of the adjacent labels, and (4) computing the frequencies of adjacent labels. This approach is based on the following idea. If the

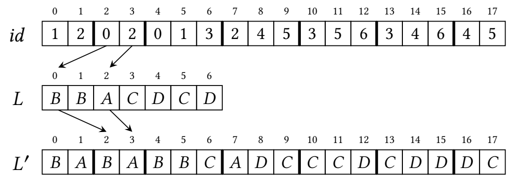


Figure 3: Gathering adjacent labels. The *i*th element of the array *L* denotes the label of vertex v_i.

adjacent labels of each vertex is sorted, the same adjacent label of one vertex is stored contiguously. This means that, if we have the indices where the adjacent labels change (called *boundaries*), the occurrence frequencies can be easily computed by the differences of boundaries, without a specific data structure such as an associative array, which is difficult to be efficiently implemented on GPUs. In other words, the counting can be implemented with simple operations on arrays, which are suitable for GPUs. The following details the four steps.

Gathering adjacent labels. The first step generates an array L' of adjacent labels by using the adjacent-vertex array *id* and the label array *L*. Concretely, this step carries out the computations $L'[i] \leftarrow L[id[i]]$ for $i \in \{0, 1, \ldots, n-1\}$, and this pattern is equivalent to the gather primitive. Thus this step is easily implemented by using gather (Line 5). Figure 3 shows an example of this step. Note that the labels in this figure are different from Figure 1 for the sake of explanation.

Sorting adjacent labels per vertex. This step sorts the array L' of adjacent labels in a per-vertex manner. This means that each subarray of L' regarding one vertex is separately sorted. A naïve way is to sort one subarray by one thread block. However, as previously mentioned, this scheme suffers from severe load imbalance because of the skewness of degree distributions. Instead, we use a data-parallel primitive, segmented sort, whose performance is not largely affected by the skewness. Specifically, the arrays L' and *ptr* are passed to segmented sort as input (Line 6).

Extracting boundaries. The third step (Lines 7–25) extracts the boundaries of adjacent labels from the array L' sorted in the previous step. This step is further divided into three sub-steps: (1) checking the boundaries of adjacent labels, (2) computing offsets for creating boundary arrays, and (3) extracting the boundaries. Figure 4 illustrates an example of these sub-steps. The goal here is to construct the two arrays *S* and *Sptr*. These arrays play a similar role to arrays in the CSR format: the first array *S* contains the information of boundaries, which are the indices where labels change, while the second array *Sptr* maintains the offsets where the boundaries for a specific vertex begin in the array *S*.

The first sub-step generates an array *F* of flags to indicate the indices where adjacent labels change (Lines 7–16). This computation is easily parallelized by assigning threads to elements. The *i*th thread takes the *i*th and $(i+1)$th elements of L', and judges whether the two labels are different or not (Line 8). If they are different, the thread stores '1' to $F[i]$ (Line 9); otherwise '0' is stored (Line 11). Meanwhile, even if the two labels are same, adjacent labels from different vertices are discriminated. Thus '1' is stored to *F* in such cases (Line 15). For example, although $L'[3] = L'[4]$ in Figure 4,

$F[3]$ is computed as '1' because $L'[3]$ is an adjacent label of v_1 and $L'[4]$ is an adjacent label of v_2.

The second sub-step computes offsets for creating boundary arrays in the next sub-step. This computation can be easily implemented by using the scan primitive. The inputs are the array F, the addition operator +, and 0 as the identity element (Line 17). The array F' in Figure 4 is the result of this primitive.

With the offsets, the arrays S and *Sptr* are constructed (Lines 18–25). Similarly to the first sub-step, the ith thread compares the ith and $(i + 1)$th elements of F' (Line 19). If they are different, the thread output boundary information (Line 20); otherwise, it just exits without outputs. Meanwhile, the array *Sptr* is also computed to maintain the correspondence between boundaries and vertices (Lines 23–25).

Computing the occurrence frequencies. This step calculates the array W of frequencies from the array S (Lines 26–29). Each element of W is computed by one thread: an element of W is the difference between two adjacent elements of S; that is, $W[i] \leftarrow S[i] - S[i-1]$ (Line 28). For the boundary case (i.e., the computation of $W[0]$), $S[0] + 1$ becomes the frequency (Line 29).

Updating labels. By using the arrays W and *Sptr*, vertex labels are updated on the GPU (Lines 30–33). To this end, we need to compute the majority of adjacent labels per vertex, from the array W of frequencies. This computation can be implemented by using the reduce primitive. Reduce with the max operator, which takes two operands and returns the larger one, is applied to each subarray of W in a per-vertex manner. Then the result is the maximum frequency for each vertex. However, this parallelization may suffer from severe load imbalance because of the skewness of degree distributions. To alleviate this issue, the proposed approach employs the segmented reduce primitive (Line 30), which is able to efficiently handle skewed data. Originally, segmented reduce only returns the array of reduced values (i.e., frequencies), while we need the labels with the maximum frequencies for updating labels. Thus, we extend the primitive so that it also returns an array I of indices corresponding to the reduced values. By using this array, we can update the labels by computing $L[i] \leftarrow L'[S[I[i]]]$ in parallel (Line 32).

3.3 Handling large-scale data

So far, we have assumed that entire graph data resides in the GPU global memory. However, this assumption does not always hold, because real-world graphs tend to have billions of edges. Our implementation requires a space cost of roughly $4n + 10m$, where n is the number of vertices and m is the number of edges. Let us consider the case that Titan X (Pascal, 12 GB memory) are employed and 32-bit integers are used for representing both graph data and labels. In this case, if we use the orkut dataset, which has 3 million vertices and 117 million edges (Section 4.2), our implementation requires approximately 4.7 GB, and this dataset can be processed entirely on the GPU. On the other hand, if we use the larger dataset called friendster [15], which has 65 million vertices and 1.8 billion edges, our implementation requires approximately 73 GB, and it is impossible to process the dataset on the GPU. Thus we need to develop an algorithm to handle large-scale graphs that cannot fit into the GPU memory.

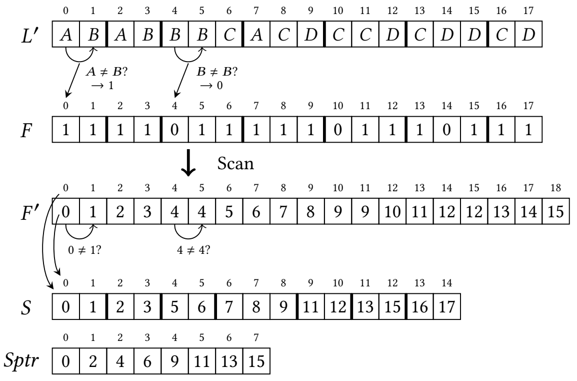


Figure 4: Extracting boundaries.

A common solution is to divide the data on the CPU, so that partitions can be handled separately by transferring them to the GPU one by one. The important points here are which data should be partitioned and which data should *not* be partitioned. In our case, the dominant factor of the space cost is edge-related data (i.e., the 'm' term), because real-world large-scale graphs tend to have a more than 10 times larger number of edges than vertices. Thus our method partitions the edge data and edge-sized temporary buffers, while the vertex-related data (i.e., labels) is kept on the GPU during the clustering process.

In the following, we first describe the partitioning method. Then naïve and improved out-of-core methods are introduced.

Partitioning. To partition edges, we set the GPU buffer size B as a parameter, and divide the edges to multiple batches, each of which contains the maximum possible number of edges without crossing vertex boundaries. For example, let us consider the graph in Figure 2 when $B = 7$. The first batch consists of the first seven edges (and corresponding *ptr* entries) associated to the first three vertices, while the second batch only contains six edges, because the next entry (i.e., *id*[13]) belongs to another vertex. By partitioning in this way, each batch can be transparently handled by the algorithm explained in Section 3.2.

Partitioning provides two additional advantages: (1) reduction of time complexity and (2) semi-synchronous updates of labels.

Complexity. Our graph-clustering algorithm uses the segmented sort primitive, which employs the merge-sort algorithm internally [4]. This means that the time complexity of sort is $O(m \log m)$ for sorting the L' array, and the entire complexity of label propagation per iteration is dominated by this term, because the other steps require only linear work. On the other hand, with partitioning, the sort is performed on the buffer of size B. This means that the total time complexity of sort becomes $(m/B) \cdot O(B \log B) = O(m \log B) = O(m)$ (B is regarded as a constant), and the complexity of label propagation is reduced to $O(m)$.

Semi-synchronous updates. Meanwhile, semi-synchronous updates lead to faster convergence [8, 10]. Label propagation algorithms update labels basically in two ways: synchronous and asynchronous updates. Synchronous label updates mean that the label of a vertex at iteration i is updated by the adjacent labels at iteration $i - 1$, while asynchronous updates mean that updates at an

Table 1: Machine configurations.

	CPU	Memory	GPU	OS
Machine 1	Intel Xeon E5-2687W v2 (8 cores at 3.40 GHz)	128 GB (DDR3 at 1866 MHz)	Tesla K40 (12 GB memory, ECC on)	CentOS release 6.5 (Final)
Machine 2	Intel Core i7-5960X (8 cores at 3.00 GHz)	32 GB (DDR4 at 2133 MHz)	GeForce GTX Titan X (Maxwell, 12 GB memory)	Ubuntu 14.04 LTS 64 bit
Machine 3	Intel Core i7-6950X (10 cores at 3.00 GHz)	64 GB (DDR4 at 2400 MHz)	NVIDIA Titan X (Pascal, 12 GB memory)	Ubuntu 16.04 LTS 64 bit

iteration are reflected to the subsequent updates at the same iteration [22]. The semi-synchronous method is a hybrid of synchronous and asynchronous schemes: it partitions the vertices of a graph into multiple sets, and labels from different sets are updated asynchronously, while labels of the same set are updated synchronously. Our method without partitioning employs a synchronous update scheme, whose convergence is slow [16]. Partitioning enables us to accelerate convergence by allowing semi-synchronous updates.

Naïve method. On the basis of the partitioning scheme, a naïve out-of-core method handles large-scale data, by alternating transfer of a batch and label updates with the batch. More specifically, the naïve method proceeds as follows. In the beginning, the first batch is transferred to the pre-allocated GPU buffer. Then, the GPU updates the labels of vertices in the batch. Having finished the updates, the next batch is transferred to the buffer and processed on the GPU. After the final batch is processed, the method proceeds to the next iteration (i.e., the first batch again), if necessary.

The naïve method requires the space cost of $n + 8B$. This means, for example, that if n is 100 million and $B = 2^{25}$, approximately 1.37 GB of GPU memory is required, regardless of the number of edges; thus, the method can perform clustering on the friendster dataset, which cannot be handled without partitioning. However, this method has a disadvantage due to CPU–GPU data transfer. In GPU computing applications, it is common that the data transfer becomes bottleneck, because GPUs are connected through the PCI Express interface, which is relatively slower than the GPU memory. Since the naïve method performs the transfer and label updates alternately, it incurs significant data-transfer latency, which should be hidden for improving performance.

Improved method. An improved method utilizes a double buffering scheme for overlapping data transfer and computation, thereby effectively hiding the latency of data transfers. This method prepares one additional buffer of size B, and one buffer is used for computation while the other buffer becomes the destination of next batch transfer. More specifically, the improved method proceeds as follows. First, a batch is transferred to one of the two GPU buffers. Then the GPU performs the updating procedure of labels of vertices in this batch, and the CPU concurrently triggers the transfer of the next batch to the empty GPU buffer. Having finished both of the computation and data transfer, the roles of buffers are switched and the procedures are repeated until convergence.

This method requires the space cost of $n+10B$, slightly larger than that of the naïve method because of double buffers. Nevertheless, it enables us to deal with very large-scale data: when $B = 2^{25}$, it can handle a graph of roughly 2.8 billion vertices on Titan X.

CPU–GPU hybrid processing. For further acceleration, the GPU out-of-core algorithm can be extended to exploit not only the GPU but also the CPU. It is important to balance the load between the CPU and the GPU for efficient hybrid processing, because the GPU is expected to be more powerful than the CPU. To this end, we prepare a double-ended queue that contains batch IDs. The GPU consumes batches from the front of the queue, while the CPU takes batches from the rear. One CPU thread manages the GPU such as kernel invocation and data transfer, while the other CPU threads perform label updates on the CPU in parallel. In this way, the load is automatically balanced, if the batch size is not too large. The effect of batch sizes is experimentally evaluated later.

4 EXPERIMENTAL EVALUATION

This section evaluates the performance of the proposed methods through experiments. Section 4.1 introduces the experimental setup. Section 4.2 shows performance results on real-world datasets, and Section 4.3 presents accuracy results using synthetic datasets.

4.1 Experimental setup

The proposed methods were implemented using CUDA and OpenMP. For comparison, the proposed methods (GPU-WOP, GPU-NO, GPU-O, and Hybrid) as well as other three implementations were used:

- *GPU-WOP* (without partitioning): A GPU implementation that is only able to handle graphs fitting into the GPU memory.
- *GPU-NO* (non-overlapping): The naïve partitioning method.
- *GPU-O* (overlapping): The improved partitioning method.
- *Hybrid*: The CPU–GPU hybrid method. The GPU uses the improved method as with GPU-O, while the CPU asynchronously updates the labels of vertices within each batch in parallel.
- *GPU-LI* (load-imbalanced): A GPU implementation does not take into account load balancing, without using the segmented sort and segmented reduce primitives. Instead, GPU-LI parallelize these computations in a block-per-vertex manner. For the sorting, we used the CUB library [17], which provides segmented radix sort functions that sort segments by assigning one block to one segment. The reduction was implemented by ourselves. This implementation is included to evaluate the impact of load balancing, and can be regarded as a comparable one to Soman and Narang [23].
- *CPU*: A parallel CPU implementation of label propagation, developed by Staudt and Meyerhenke [24]. The implementation is available as a part of open-source library *NetworKit*.[2]

[2]https://networkit.iti.kit.edu/index.html

Table 2: Real-world datasets.

Dataset	n	m	Type
amazon [15]	334,863	925,872	Social network
dblp [15]	317,080	1,049,866	Social network
youtube [15]	1,134,890	2,987,624	Social network
livejournal [15]	3,997,962	34,681,189	Social network
orkut [15]	3,072,441	117,185,083	Social network
friendster [15]	65,608,366	1,806,067,135	Social network
uk-2007-05 [3]	105,896,555	3,301,876,564	Web graph

The termination condition for label propagation was set as the number of updated labels in one iteration is less than $10^{-5}n$ or the number of iterations reaches ten, where n is the number of vertices.

We used three GPU-equipped machines, whose configurations are summarized in Table 1. In the following experiments, Machine 3 (i.e., NVIDIA Titan X Pascal) was basically used because it has the latest GPU among them. The other machines are employed for comparing the performance with different GPUs. For running the CPU implementation, we used Machine 1 because it provides the fastest result. In this case, 16 threads were used because the CPU has 8 cores and supports up to 16 threads by Hyper-Threading.

4.2 Results on real-world datasets

This subsection evaluates the performance of the proposed methods by using multiple real-world datasets. The datasets were obtained from Stanford Network Analysis Project [15] and Laboratory for Web Algorithmics [3]. Table 2 summarizes the datasets used in the experiments. The experiments measured elapsed time from when graph data is ready on the main memory until the clustering result is placed on the main memory. Note that the data transfer time between the CPU and the GPU was included in the measurement.

Small-scale datasets. We first use three small-scale datasets (i.e., amazon, dblp, and youtube) to compare CPU, GPU-WOP, and GPU-LI. Figure 5 shows the timing results, where GPU-WOP and GPU-LI were run on NVIDIA Titan X (Pascal). From the figures, we can make two observations: (1) GPU-WOP is the fastest method, up to 14.3 times faster than CPU; and (2) GPU-LI exhibits significantly worse performance than GPU-WOP.

Observation 1. Overall, GPU-WOP is the fastest method, and the CPU parallel implementation is the second fastest; GPU-WOP outperforms CPU by factors of 4.24 to 14.3. These results reveal that our GPU implementations have superior performance over an existing parallel implementation.

Observation 2. The second observation implies the importance of load balancing as well as resource utilization. In particular, the results on youtube is the worst case where GPU-LI is 35 times as slow as GPU-WOP. The youtube dataset has a highly skewed degree distribution with the average degree of 2.63 and the maximum degree of 28,754. Since GPU-LI assigns work of a vertex to one block, it not only fails to achieve load balancing, but also underutilizes the compute resources because blocks deal with only 2.63 vertices on average.

Middle-scale datasets. We next use two middle-scale datasets, livejournal and orkut, to examine the performance of CPU, GPU-WOP, GPU-NO, GPU-O, and Hybrid. Although these datasets fit into the GPU memory, the partitioning-based methods are also included in order to inspect performance differences between with and without partitioning. Note also that Hybrid used 20 CPU threads according to the number of supported threads on Machine 3. Figure 6 illustrates the results, where GPU-NO, GPU-O, and Hybrid uses the buffer sizes resulting in the best performance: the buffer sizes on livejournal are 2^{26}, 2^{22}, and 2^{22} for GPU-NO, GPU-O, and Hybrid, respectively; and on orkut 2^{25}, 2^{24}, and 2^{23} are used for GPU-NO, GPU-O, and Hybrid, respectively. The performance with different buffer sizes will be separately discussed later.

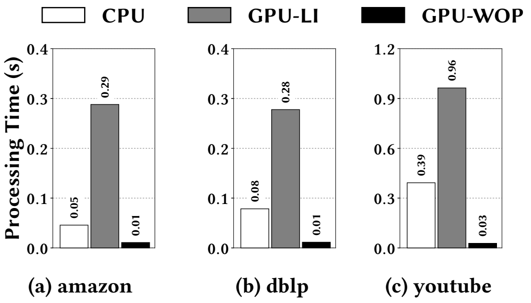


Figure 5: Results on small-scale datasets.

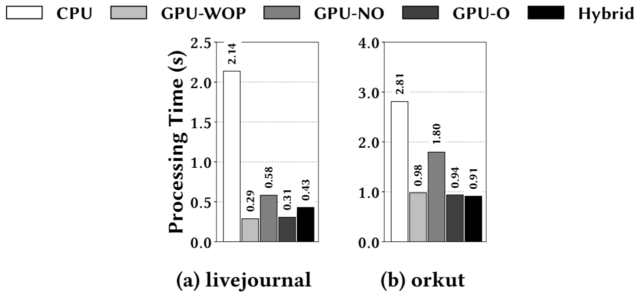


Figure 6: Results on middle-scale datasets.

The following observations can be made from the figures: (1) GPU-based implementations outperform CPU by a factor of up to 7.4 and 3.1 on livejournal and orkut, respectively; (2) GPU-O enables us to obtain almost double performance gains over GPU-NO; (3) on orkut, GPU-O outperforms GPU-WOP; and (4) Hybrid is slightly faster than GPU-O on orkut, while it is slower on livejournal.

Observation 1. GPU-WOP is fastest on livejournal, 7.4 times faster than CPU. On the other hand, Hybrid is fastest for orkut, and the performance improvement over CPU is up to 3.1, which is lower than the case of livejournal. This is due to dataset characteristics, because the CPU implementation reduces the computational cost by filtering *inactive* vertices, while our implementations process all vertices in a bulk-synchronous way. A vertex is inactive in an iteration if all the labels of its neighbors have not changed from the previous iteration [24]. Since the computation performed for the inactive vertex in the current iteration is the same as that of the previous iteration, such computations need not be performed. In the experiments, orkut generates more inactive vertices than livejournal, and CPU avoids the computations for inactive vertices, while our implementations carry out them.

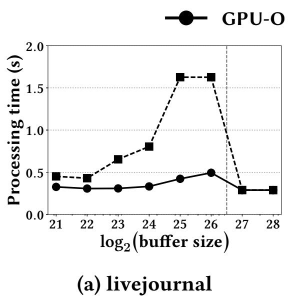


Figure 7: Varying buffer sizes on middle-scale datasets.

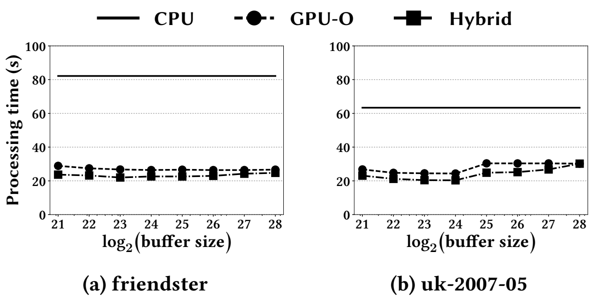


Figure 8: Results on large-scale datasets.

Observation 2. The second observation is concerned with the difference between the overlapping and non-overlapping approaches. As the figure shows, GPU-O is roughly two times faster than GPU-NO. Moreover, GPU-NO is up to two times slower than GPU-WOP. This is because GPU-NO transfers batches in a non-overlapping fashion at every iteration. As a result, the communication incurs significant overhead and becomes the bottleneck. On the other hand, the results show that GPU-O can effectively hide the communication cost by overlapping computation and data transfers.

Observation 3. The third observation implies that partitioning is beneficial for performance. Recall that GPU-WOP transfers all necessary data to the GPU and perform clustering without further communication, while GPU-O partitions the graph data and perform clustering by overlapping computation and communication. The performance improvement is mainly due to two reasons: (1) shorter transfer time of the initial copy and (2) the reduction of time complexity. Before clustering starts, GPU-WOP transfers all data, whereas GPU-O copy a smaller batch of buffer size B. Thus, GPU-O finishes the initial copy faster than GPU-WOP, and consequently GPU-O is able to start clustering earlier. The second reason is the point discussed in Section 3.3: GPU-WOP requires the time complexity of $O(m \log m)$, whereas GPU-O has the time complexity of $O(m)$, where m is the number of edges. Therefore GPU-O is more scalable than GPU-WOP with regard to m. As for livejournal, GPU-O is slightly slower than GPU-WOP; this is because the dataset is relatively small, and the overhead due to partitioning becomes more apparent.

Observation 4. The fourth observation comes from the difference of dataset sizes. Since the livejournal dataset is not quite large, when it is divided into batches, the total number of batches is small. As a result, load balancing between the CPU and the GPU cannot be well achieved. On the other hand, orkut is relatively large, about 3.44 times larger than livejournal. Thus it produces a sufficient number of batches, and the hybrid processing effectively works.

Varying buffer sizes. We next investigate the performance changes with varying buffer sizes. Figure 7 plots the processing time of GPU-O and Hybrid as a function of buffer size. The gray dashed vertical line indicates the turning point from with-partitioning to without-partitioning cases; e.g., a buffer of size 2^{28} accommodates the datasets entirely. In such cases, there is only one batch, and GPU-O and Hybrid become effectively the same as GPU-NO. That is the reason why the two implementations with large buffer sizes exhibit the same performance.

Table 3: Time breakdown of GPU-O when $B = 2^{24}$.

Kernel	friendster	uk-2007-05
`segmented_sort`	4.26 s (16%)	9.21 s (39%)
`gather`	18.5 s (69%)	1.60 s (6.7%)

The performance of GPU-O changes like a convex function until the turning point as the buffer size increases. This behavior is due to the throughput of segmented sort. As the buffer size increases, the sort throughput varies like a concave function; i.e., the throughput increases up to a certain buffer size, and then decreases for larger buffer sizes. In Figure 7, GPU-O achieves the best performance at around 2^{24}, where the sort throughput reaches the maximum. As for Hybird, smaller buffer sizes (2^{22}–2^{23}) result in the best. This is due to load balancing between the CPU and the GPU, as previously mentioned. In fact, large buffer sizes (e.g., 2^{26}) lead to significantly worse performance, because there are only a small number of huge batches.

Large-scale datasets. We here see the results on two large-scale datasets, friendster and uk-2007-05, to compare the performance of CPU, GPU-O, and Hybrid. Figure 8 depicts the processing time as a function of buffer size. From the figures, we can make three observations: (1) Hybrid is consistently faster than GPU-O and CPU; (2) the GPU-based implementations result in similar performance for both datasets, even though their numbers of edges are largely different; and (3) moreover, CPU finishes faster on uk-2007-05 than on friendster.

Observations 1–3. On friendster (Figure 8(a)), Hybrid achieves a speedup of up to 3.74 over CPU and 1.2 over GPU-O. On the other hand, Hybrid attains a slightly smaller speedup of 3.1 over CPU on uk-2007-05. This behavior, including the above second and third observations, is probably due to the vertex ordering [1] of the datasets (i.e, how the vertices are numbered). If neighboring vertices have similar vertex indices, caches can be exploited because such vertices are frequently accessed simultaneously. To investigate this point, we inspect the time breakdown of GPU-O for both datasets. Table 3 summarizes the breakdown, showing only the most important kernels. The table reveals that the bottleneck is `gather` on friendster, whereas `segmented_sort` on uk-2007-05. The performance of `gather` is highly dependent on the vertex ordering, while `segmented_sort` little depends. Thus it can be stated that the vertex ordering of uk-2007-05 is memory-friendly, and the ordering of friendster is not. This issue can be resolved by applying

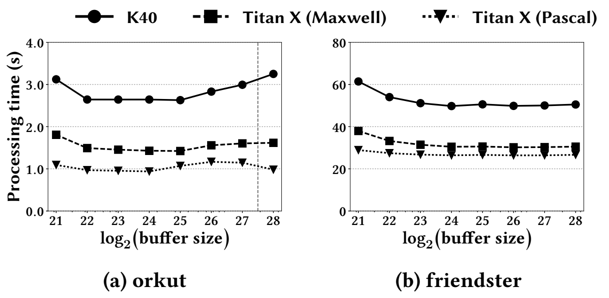


Figure 9: Results with different GPUs on two datasets.

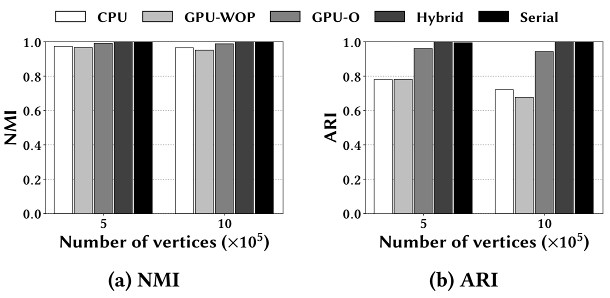


Figure 10: Results on synthetic datasets.

a technique called *graph reordering* [1], but it is beyond the scope of this paper.

Different GPUs. Finally, we discuss results of GPU-O with different GPUs. Figure 9 shows the results on orkut and friendster, the two largest datasets that can be processed by all of the machines. Note that Machine 2 cannot process uk-2007-05 because of the limitation of *not GPU memory but CPU memory.*

Basically, newer GPUs provides better performance. Specifically, Titan X (Pascal) is up to 2.80 times and 1.89 times faster than K40 on orkut and friendster, respectively. Compared to Titan X (Maxwell), Pascal achieves speedups of 1.52 and 1.14 on the two datasets, respectively. These results are attributed to not only the advances of available parallelism and memory bandwidth, but also several microarchitectural differences. Important ones among them are the increase of register-file and shared-memory capacity. For instance, Pascal-based Titan X contains two times larger register file and three time larger shared memory over Tesla K40. As a result, newer GPUs allow us to achieve better occupancy. This means that a larger number of threads can be scheduled concurrently on GPUs, and latency hiding mechanism more effectively works. Therefore, newer GPUs lead to superior performance over older ones.

4.3 Results on synthetic datasets

This subsection evaluates the clustering accuracy by using synthetic datasets. Datasets are generated by the *LFR (Lancichinetti–Fortunato–Radicchi)* benchmark [14], which is commonly employed to evaluate graph-clustering algorithms. The LFR benchmark generates graph data with community information. This information is compared with the partition generated by an algorithm for evaluating its accuracy. To this end, the following two famous metrics are adopted [7]: (1) *NMI (normalized mutual information)* and (2) *ARI (adjusted Rand index).*

NMI measures the similarity between two partitions C and C' on the basis of an information-theoretic approach:

$$\mathrm{NMI}(C, C') = \frac{-2\sum_{i,j} N_{ij}\log(N_{ij}N_t/N_{i\cdot}N_{\cdot j})}{\sum_i N_{i\cdot}\log(N_{i\cdot}/N_t) + \sum_j N_{\cdot j}\log(N_{\cdot j}/N_t)},$$

where N is the confusion matrix whose entry N_{ij} is the number of vertices that appear in both C_i and C_j, $N_{i\cdot}$ and $N_{\cdot j}$ are the sum over the row i and the sum over the column j, respectively, and N_t is the sum over the entire matrix. NMI takes values between 0 and 1, and if two partitions are same, it takes 1.

ARI is a metric based on pair counting:

$$\mathrm{ARI}(C, C') = \frac{\sum_{i,j}\binom{N_{ij}}{2} - M}{\frac{1}{2}\left[\sum_i \binom{N_{i\cdot}}{2} + \sum_j \binom{N_{\cdot j}}{2}\right] - M},$$

where N is the confusion matrix and $M = \left[\sum_i \binom{N_{i\cdot}}{2}\sum_j \binom{N_{\cdot j}}{2}\right]/\binom{n}{2}$. ARI takes 1 in the base case.

Figure 10 shows the results on synthetic datasets, where the y-axis is the average values of each metric over five trials for two different graph sizes. The data is generated with the average degree of 20, the maximum degree of $0.01n$, where n is the number of vertices, and the mixing parameter μ of 0.2. The "serial" bar means the results of a serial implementation of label propagation with asynchronous label updates.

Overall, serial achieves the most accurate results, and Hybrid is the next. While GPU-WOP results in the worst accuracy, owing to the synchronous label updates, GPU-O can remedy this issue by employing semi-synchronous updates. Besides, Hybrid produces more accurate results. This is because the CPU part of Hybrid employs asynchronous updates; thus it gets closer to the convergence than the GPU-only implementations in the same number of iterations. The results demonstrate that our method is able to achieve high performance without significant loss of accuracy.

5 RELATED WORK

Graph clustering has recently been paid significant attention, and many algorithms and measures to evaluate clustering results have been proposed [12, 26]. This section gives a brief survey of graph clustering algorithms and their parallelization.

Serial graph clustering. *Modularity* [18] is a popular measure to evaluate clustering quality, and many modularity-based algorithms have been proposed. The *Louvain* method [6] is a well-known greedy algorithm for maximizing modularity. Louvain enables fast clustering by computing modularity gains when a vertex is moved from a cluster to another. However, it is pointed out that modularity has a limitation called *resolution limit* [11], which means that, as graphs become larger, small clusters cannot be captured by modularity.

As another class of graph clustering algorithms, *label propagation* was proposed [16, 22]. Raghavan et al. [22] proposed a basic label propagation algorithm. Leung et al. [16] improved the basic label propagation algorithm by introducing two heuristics.

Wang et al. [26] conducted extensive experiments for fair comparisons of ten graph clustering algorithms, by implementing them with a common framework. Consequently, label propagation achieves higher performance and more accurate results compared with other algorithms. Therefore, our work focuses on the acceleration of label propagation.

Parallel graph clustering. Parallelization of modularity-based algorithms has been also extensively studied because of its popularity. For instance, Djidjev and Onus [9] accelerated modularity-based clustering by utilizing the property that the problem of modularity maximization can be reduced to a minimum weighted cut problem on a complete graph with the same vertices. With this reduction, modularity maximization can be efficiently solved by existing parallel graph partitioning techniques.

Parallelization of label propagation has also been proposed, because it is suitable for parallel processing as only vertex-local information is needed for clustering. Cordasco and Gargano [8] proposed a novel method that updates vertex labels *semi-synchronously* by using graph coloring. Although they did not conduct experiments with parallel environments, later Duriakova et al. [10] extended, implemented, and evaluated the semi-synchronous scheme with multi-core CPUs. Staudt and Meyerheck [24] implemented parallel label propagation as well as the Louvain method for multi-core CPUs. They introduce filtering of inactive vertices, thereby reducing the computational cost.

Compared to the above semi-synchronous scheme, our work simply divides vertices into multiple sets, but the coloring-based scheme can be easily incorporated to our methods for improving the convergence rate. Meanwhile, the implementation by Staudt and Meyerheck employs a filtering scheme, whereas our methods handle all vertices at every iteration. Thus, implementing such filtering mechanism to our methods can provide higher performance.

GPU graph clustering. GPUs have also been utilized to accelerate graph clustering. Stovall et al. [25] parallelized on GPUs the *SCAN* algorithm [27], which is based on structural similarity. Label propagation on GPUs was proposed by Soman and Narang [23]. They developed a parallel label propagation method targeting both multi-core CPUs and GPUs. However, the details of GPU implementation is not well discussed and their method does not well take into account load balancing, which is extremely important to efficiently handle real-world graphs. Our method achieves load balancing almost completely, by exploiting primitives whose performance is not largely affected by the existence of skewness.

6 CONCLUSION

This paper has presented a fast graph clustering algorithm based on parallel label propagation on the GPU. To efficiently perform graph clustering, the algorithm of label propagation is transformed into a series of data-parallel operations. Load balancing is taken into account by using the primitives that make the load among threads and blocks well balanced. In addition, we have developed GPU out-of-core implementations to resolve the issue of GPU memory. This implementation is further extended to utilize not only GPUs and but also CPUs. Experiments on both real-world and synthetic datasets show that our proposed method achieves superior performance than existing methods without sacrificing accuracy. In particular, our hybrid implementation is 3.1 times as fast as an existing parallel CPU implementation on a graph with 3.3 billion edges.

ACKNOWLEDGMENTS

This research was partly supported by the Grant-in-Aid for Scientific Research (B) (#26280037) and the Grant-in-Aid for JSPS Fellows (#15J02121) from Japan Society for the Promotion of Science. This paper is also partly based on results obtained from a project commissioned by the New Energy and Industrial Technology Development Organization (NEDO).

REFERENCES

[1] J. Arai, H. Shiokawa, T. Yamamuro, M. Onizuka, and I. Sotetsu. Rabbit Order: Just-in-Time Parallel Reordering for Fast Graph Analysis. In *Proc. IPDPS*, pp. 22–31, 2016.
[2] A.-L. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, vol. 286, no. 5439, pp. 509–512, Oct. 1999.
[3] P. Boldi, M. Santini, and S. Vigna. A Large Time-Aware Graph. *SIGIR Forum*, vol. 42, no. 2, pp. 33–38, Dec. 2008.
[4] S. Baxter. Modern GPU, ver. 1.0. https://github.com/moderngpu/moderngpu.
[5] N. Bell and M. Garland. Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors. In *Proc. SC*, pp. 18:1–18:11, 2009.
[6] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre. Fast unfolding of communities in large networks. *J. Stat. Mech. Theor. Exp.*, vol. 2008, no. 10, p. P10008, Oct. 2008.
[7] M. Chen, K. Kuzmin, and B. K. Szymanski. Community Detection via Maximization of Modularity and Its Variants. *IEEE Trans. Computational Soc. Syst.*, vol. 1, no. 1, pp. 46–65, Mar. 2014.
[8] G. Cordasco and L. Gargano. Label Propagation Algorithm: A Semi-synchronous Approach. *Int. J. Soc. Netw. Min.*, vol. 1, no. 1, pp. 3–26, 2012.
[9] H. N. Djidjev and M. Onus. Scalable and Accurate Graph Clustering and Community Structure Detection. *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 5, pp. 1022–1029, May 2013.
[10] E. Duriakova, N. Hurley, D. Ajwani, and A. Sala. Analysis of the Semi-synchronous Approach to Large-scale Parallel Community Finding. In *Proc. COSN*, pp. 51–62, 2014.
[11] S. Fortunato and M. Barthélemy. Resolution limit in community detection. *Proc. Natl. Acad. Sci.*, vol. 104, no. 1, pp. 36–41, Dec. 2007.
[12] S. Fortunato. Community detection in graphs. *Phys. Rep.*, vol. 486, no. 3–5, pp. 75–174, Feb. 2010.
[13] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational Query Coprocessing on Graphics Processors. *ACM Trans. Database Syst.*, vol. 34, pp. 21:1–21:39, Dec. 2009.
[14] A. Lancichinetti, S. Fortunato, and F. Radicchi. Benchmark graphs for testing community detection algorithms. *Phys. Rev. E*, vol. 78, no. 4, p. 046110, Oct. 2008.
[15] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, Jun. 2014.
[16] I. X. Y. Leung, P. Hui, P. Liò, and J. Crowcroft. Towards real-time community detection in large networks. *Phys. Rev. E*, vol. 79, p. 066107, Jun 2009.
[17] D. Merrill. CUB, ver. 1.6.4. http://nvlabs.github.io/cub/.
[18] M. E. J. Newman and M. Girvan. Finding and evaluating community structure in networks. *Phys. Rev. E*, vol. 69, no. 2, p. 026113, Feb. 2004.
[19] NVIDIA. CUDA C Programming Guide. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.
[20] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. GPU Computing. *Proc. IEEE*, vol. 96, no. 5, pp. 879–899, May 2008.
[21] S. Papadopoulos, Y. Kompatsiaris, A. Vakali, and P. Spyridonos. Community Detection in Social Media. *Data Min. Knowl. Discov.*, vol. 24, no. 3, pp. 515–554, May 2012.
[22] U. N. Raghavan, R. Albert, and S. Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Phys. Rev. E*, vol. 76, p. 036106, Sep. 2007.
[23] J. Soman and A. Narang. Fast Community Detection Algorithm with GPUs and Multicore Architectures. In *Proc. IPDPS*, pp. 568–579, 2011.
[24] C. L. Staudt and H. Meyerhenke. Engineering Parallel Algorithms for Community Detection in Massive Networks. *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 1, pp. 171–184, Jan. 2016.
[25] T. R. Stovall, S. Kockara, and R. Avci. GPUSCAN: GPU-Based Parallel Structural Clustering Algorithm for Networks. *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 12, pp. 3381–3393, Dec. 2015.
[26] M. Wang, C. Wang, J. X. Yu, and J. Zhang. Community Detection in Social Networks: An In-depth Benchmarking Study with a Procedure-oriented Framework. *Proc. VLDB Endow.*, vol. 8, no. 10, pp. 998–1009, Jun. 2015.
[27] X. Xu, N. Yuruk, Z. Feng, and T. A. J. Schweiger. SCAN: A Structural Clustering Algorithm for Networks. In *Proc. KDD*, pp. 824–833, 2007.