IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS

Clip: A Disk I/O Focused Parallel Out-of-core Graph Processing System

Zhiyuan Ai*, Mingxing Zhang*, Yongwei Wu, *Senior Member, IEEE*, Xuehai Qian, Kang Chen, and Weimin Zheng, *Senior Member, IEEE*

Abstract—Existing parallel out-of-core graph processing systems focus on improving disk I/O locality, which leads to restrictions on their programming models. Although improving the locality, these constraints also restrict the expressiveness and hence only sub-optimal algorithms are supported. These sub-optimal algorithms typically incur *sequential*, but *much larger*, amount of disk I/O. In this paper, we explore a fundamentally different tradeoff: less total amount of I/O rather than better locality. We show that out-of-core graph processing systems uniquely provide the opportunities to lift the restrictions of the programming model in a feasible manner. To demonstrate the ideas, we build Clip, which enables more efficient algorithms that require much less amount of total disk I/O. Our experiments show that the algorithms that can be only implemented in Clip are much faster than the original disk-locality-optimized algorithms.
We also further extend our technique's scope of application by providing a semi-external mode. Our analysis and evaluation demonstrate that semi-external is not only feasible for many cases, but also be able to deliver a significant speedup for important graph applications. Moreover, we further improve the performance of originally supported applications by designing more optimizations and evaluate our system on NVMe SSD.

Index Terms—Graph Processing, Parallelization, Big Data, Disk I/O.

1 Introduction

As an alternative to distributed graph processing, single-machine parallel out-of-core graph processing systems can largely eliminate all the challenges of using a distributed framework. Since the ease of use, several out-of-core systems have been developed recently [2], [3], [4], [5]. These systems make practical large-scale graph processing available to anyone with a modern PC. It is also demonstrated that the performance of a single ordinary PC running GridGraph is competitive with a distributed graph processing framework using hundreds of cores [4].

The major performance bottleneck of out-of-core systems is disk I/O. Therefore, improving the locality of disk I/O has been the main optimization goal. The current systems [2], [3], [4], [5] use two requirements to achieve this goal. First, the execution engine defines a specific processing order for the graph data and only iterates the edges/vertices according to such order, which means that each edge/vertex is processed *at most once* in an iteration. By avoiding fully asynchronous execution, this technique naturally reduces the tremendous amount of random disk I/O. The second is the *neighborhood constraint* that requires the user-defined function to only access neighborhood of its corresponding input vertex/edge. This requirement improves the locality of disk I/O and also makes automatic parallelization of in-memory processing practical.

According to our investigation, almost all existing out-of-core systems enforce the above two requirements in their programming and execution model, which assure the good disk I/O locality for their supported algorithms. However, these restrictions (e.g., process each loaded block at most once, neighborhood constraint) also affect the models' expressiveness and flexibility and lead to the sub-optimal algorithms. As a result, the execution incurs *sequential, but excessive*, the amount of disk I/O, compared with more efficient algorithms which require drastically less iterations.

In fact, the "at most once" requirement wastes the precious disk bandwidth. Many graph algorithms (e.g. SSSP, BFS) are based on iterative improvement methods and can benefit from iterating multiple times on a loaded data block. Moreover, many important graph problems (e.g., WCC, MIS) can be solved with much less iterations (typically only **one** pass is enough) by changing algorithms. However, these algorithms require the removal of "neighborhood constraint". In essence, we argue that the current systems follow a wrong trade-off: they improve the disk I/O locality at the expense of less efficient algorithms with the larger amount of disk I/O, wasting the precious disk bandwidth. As a consequence, current out-of-core systems only achieve *sub-optimal* performance.

In this paper, we propose Clip, a novel parallel out-of-core graph processing system, in which supporting more efficient algorithms is the primary concern. We argue that the out-of-core systems uniquely provide the opportunities to lift the restrictions of the programming and execution model (process each loaded block at most once, neighborhood constraint) in a feasible manner. Specifically, Clip is designed with the principle of "squeezing out all the value of loaded data". It defines a programming model that supports *1)* **loaded data reentry** by allowing more flexible processing order; and *2)* **beyond-neighborhood** by allowing an "edge function" to update vertex properties that do not

*Z. Ai and M. Zhang equally contributed to this work.

- *An earlier version of this work [1] appeared in ATC 2017.*
- *Zhiyuan Ai, Yongwei Wu, Kang Chen, Weimin Zheng are with the Department of Computer Science and Technology, Tsinghua National Laboratory for Information Science and Technology (TNLIST), Tsinghua University, Beijing 100084, China, Research Institute of Tsinghua University in Shenzhen, Shenzhen 518057, China. Email: {azy13@mails, wuyw@, chenkang@, zwm-dcs@}tsinghua.edu.cn.*
- *Mingxing Zhang is now with Sangfor Inc. Email: zhang.mingxing@outlook.com*
- *Xuehai Qian is currently with the University of Southern California, USA. Email: xuehai.qian@usc.edu*

Digital Object Identifier 10.1109/TPDS.2018.2858250

belong to the edge's neighborhood.

Essentially, CLIP chooses an alternative trade-off by enabling more efficient algorithms and more flexible executions at the expense of accessing vertices beyond the neighborhood. Obviously, randomly accessing vertices in disk incurs random disk I/O that is detrimental to performance. To mitigate this issue, CLIP simply *mmap* all the vertex data into memory. Using this method, although the vertex data could reside in either memory or disk, Lin et al. [6] showed that the built-in caching mechanism of mmap is particularly desirable for processing real-world graphs, which often exhibit power-law degree distributions [7].

Moreover, we further explore CLIPs scope of applicability and design some more optimizing techniques and application algorithms. Specifically, we first show that CLIP can give full play to the advantages of semi-external mode for some complex graph applications which require a random accesses of vertices to achieve their best performance. Second, we propose a novel diagonal-based partitioning and scheduling mechanism, which can further reduce the number of iterations.

Meanwhile, we further consider a new environment of faster storage media (NVMe SSD). Our experiment results demonstrate that, if NVMe SSD (2.88GB/s for sequential read) is used, the CPU consumption of CLIP can become the new bottleneck. Therefore, to meet this new challengewe first new-design the parallel algorithms for beyond-neighborhood applications by using the flexible programming model of CLIP. Second, we propose two efficient and novel selective scheduling mechanisms, which significantly improve the performance of loaded data reentry applications in memory mode.

The evaluation of our system consists of three parts. First, we evaluate the effectiveness of loaded data reentry and beyond-neighborhood. According to our experiments, both of these two technologies can significantly reduce the number of required iterations(even reduced to one iteration). Therefore, CLIP can achieve up to $14.06\times$ speedup for intrinsically iterative algorithms(like SSSP, BFS) and $3.25\times$-$4264\times$ speedup for WCC and MIS algorithms.

Second, we evaluate our semi-external mode and diagonal-based partitioning and scheduling mechanism. Results show that, the performance of CLIP's semi-external mode can largely surpass ($12.63\times$-$195.1\times$) existing works on MCST and Coloring algorithms. And the diagonal-based partitioning and scheduling mechanism can further achieve $1.17\times$-$1.5\times$ speedup on SSSP algorithm.

Finally, we evaluate our new-designed parallel algorithms and the novel selective scheduling mechanism. Without considering loading time, the new-designed parallel algorithms can achieve up to $8.91\times$ speedup for WCC and MIS algorithms (16 threads) and the novel selective scheduling mechanism can achieve up to $43.4\times$ speedup for BFS algorithm. Furthermore, we evaluate our system on NVMe SSD and compare it with the state-of-the art system MOSAIC [8]. Results show that, thanks to the significant reduction in the number of iterations, CLIP can also achieve a significant speedup over MOSAIC on WCC (up to $1514\times$) and BFS (up to $7.21\times$). Moreover, our optimization strategies can further improve the performance of our original system(up to $2.88\times$ speedup).

2 OUT-OF-CORE GRAPH PROCESSING

GraphChi [2] is the first large-scale out-of-core graph processing system that supports vertex programs. In GraphChi, the whole set of vertices are partitioned into "intervals", and the system only processes the related sub-graph of an interval at a time (i.e., only the edges related to vertices in this interval are accessed). This computation locality of vertex program (i.e. access only the neighborhood of input vertex) makes it easy for GraphChi to reduce random disk accesses. As a result, GraphChi requires a small number of non-sequential disk accesses and provides competitive performance compared to a distributed graph system [2].

Some successor systems (e.g., X-Stream [3], GridGraph [4]) propose an edge-centric programming model to replace the vertex-centric model used in GraphChi. A user-defined function in the edge-centric model is only allowed to access the data of an edge and the related source and destination vertices. This requirement also enforces a similar neighborhood constraint as the vertex-centric models, and hence ensures the systems to incur only limited amount of random disk I/O.

However, although these existing out-of-core graph processing systems differ vastly in detailed implementation, they share two common design patterns: *1).* Graph data (i.e. edges/vertices) is always (selectively) loaded in specific order and each of the loaded data block is processed at most *once* in an iteration; *2).* They all require that the user-defined functions should only access the *neighborhood* of the corresponding edge/vertex.

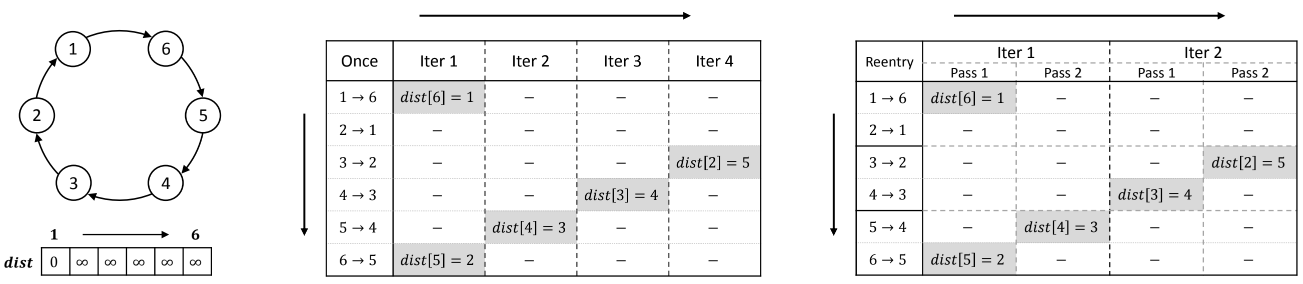
3 REDUCING DISK I/O

According to our investigation, these two shared patterns could potentially prohibit programmers from constructing more efficient algorithms, and therefore increase the total amount of disk I/O. Motivated by this observation, our approach lifts the restrictions in the current programming and execution model by: *1)* providing more flexible processing order; and *2)* allowing the user-defined function to access an arbitrary vertex's property. This section discuss the rationale behind these two common patterns, and why they are *not* always necessary in an out-of-core system. More importantly, with the restrictions removed, how our approach could enable more efficient algorithms that require less number of iterations and less amount of disk I/O. In essence, our approach *squeezes out all the values of loaded data*.

3.1 Reentry of Loaded Data

Out-of-core systems typically define a specific processing order for the graph data and only iterate the edges/vertices according to such order. This is because a fully asynchronous graph processing would incur a lot of *random accesses* to the graph data, drastically reducing disk I/O performance. However, this strategy could potentially increase the number of required iterations of many iterative improvement algorithms (e.g. SSSP, BFS).

Figure 1 shows an example that calculates single source shortest path (SSSP) on a graph of 6 vertices. In SSSP, the vertex property $dist[v]$ is initialized to 0 for vertex 1 and ∞ for the others (Figure 1 (a)). The edge function applied to each edge (u, v) checks whether $dist[v]$ is larger than $dist[u] + 1$, If it is true, $dist[v]$ is *immediately* updated as



Once	Iter 1	Iter 2	Iter 3	Iter 4
1 → 6	$dist[6] = 1$	–	–	–
2 → 1	–	–	–	–
3 → 2	–	–	–	$dist[2] = 5$
4 → 3	–	–	$dist[3] = 4$	–
5 → 4	–	$dist[4] = 3$	–	–
6 → 5	$dist[5] = 2$	–	–	–

Reentry	Iter 1		Iter 2	
	Pass 1	Pass 2	Pass 1	Pass 2
1 → 6	$dist[6] = 1$	–	–	–
2 → 1	–	–	–	–
3 → 2	–	–	–	$dist[2] = 5$
4 → 3	–	–	$dist[3] = 4$	–
5 → 4	–	$dist[4] = 3$	–	–
6 → 5	$dist[5] = 2$	–	–	–

(a) An example graph (b) The loaded block is processed only once (c) The loaded block is processed multiple times

Fig. 1: SSSP example. All the edges of this graph have the same distance set to 1.

$dist[u] + 1$. Figure 1 (b) shows the execution, where each iteration *sequentially* loads *one edge at a time*, processes it and updates $dist[v]$ if necessary. As a result, 4 iterations are needed. The number of iterations is determined by the diameter of the graph. To mitigate this issue, some prior systems (e.g., GraphChi, GridGraph) *1)* allows an update function to use the most recent values of the edges/vertices; and *2)* provides selective scheduling mechanisms that skip certain data blocks if they are not needed. Although these optimizations enable "asynchronous execution", the essential workflow is not changed as each block loaded is still processed *at most once* in every iteration.

However, the current approaches *fail to exhaust the value of loaded data*, because a block of edges rather than only one edge is loaded at a time. While the edges in a block are independent, they constitute a sub-graph in which information could be propagated by processing it multiple times. In another word, the system could squeeze more value of the loaded data. This approach is a mid-point between fully synchronous and asynchronous processing and achieves the best of both: ensuring sequential disk I/O by synchronously processing between blocks and, at the same time, enabling asynchronous processing within each block.

The idea is illustrated in the example in Figure 1 (c). Here, we partition the edges into blocks that each contains two edges, and we apply two computation passes to every loaded block. As a result, the number of iterations is reduced to 2. We call the proposed simple optimization technique *loaded data reentry*. As we see from the SSSP example in Figure 1, loaded data reentry could effectively reduce the number of iterations, reduce the amount of disk I/O and eventually reduce the whole execution time. For each loaded data block, more CPU computation is required. Considering the relative speed of CPU and disk I/O, trading CPU computation for less disk I/O is certainly a sensible choice.

3.2 Beyond the Neighborhood

"Loaded data reentry" is simple and requires only moderate modifications to be applied to existing systems (e.g., GridGraph). However, to apply the principle of "squeezing all the values of loaded data" to more applications, we found that the **neighborhood constraint** imposed by existing systems prohibits the possibility of optimizing in many cases. This neighborhood constraint is enforced by almost all single-machine graph processing systems because in this way one can easily infer the region of data that will be modified by the inputs, which is necessary for disk I/O optimizations. Despite the rationale behind, neighborhood constraint limits the expressiveness of programming model in a way that certain algorithms cannot be implemented in the most efficient manner.

We use weakly connected component (WCC) to explain the problem. WCC is a popular graph problem that calculates whether two arbitrary vertices in a graph are *weakly connected* (i.e., connected after replacing all the directed edges with undirected edges). With the existing programming models, this problem can only be solved by a label-propagation-based algorithm, in which each node repeatedly propagates its current label to its neighbors and update itself if it receives a lower label. The intrinsic property of this algorithm (i.e., the label information only propagates one hop in each iteration) inevitably causes the large number of required iterations to coverage, especially for graphs with large diameters. However, if the user-defined function is allowed to update the property of an arbitrary vertex, a disjoint-set [9], [10] data structure can be built in memory. Based on the disjoint-set, WCC problem for any graph can be solved with only *one* pass of the edges.

In general, this method is used in a class of graph algorithms termed *Graph Stream Algorithms* [11], where a graph $G = (V, E)$ is represented as a stream of edges, the storage space of an algorithm is bounded by $O(|V|)$. Graph Stream Algorithms has been studied by the theoretical community for about twenty years [11], [12], and it has been shown that if a randomly accessible $O(|V|)$ space is given, many important graph algorithms can be solved by reading only one (or a few) pass(es) of the graph stream [13]. Unfortunately, the whole class of Graph Stream Algorithms cannot be implemented by the programming model of current disk-based out-of-core systems (or only in a very inefficient manner).

4 CLIP

To support the loaded data reentry and beyond-neighborhood optimization, we design and implement a C++-based novel out-of-core graph processing system, CLIP. CLIP allows users to flexibly write more efficient algorithms that require less number of iterations (and less disk I/O) than algorithms based on previous programming models. The flexibility of our system is achieved due to *1)* its unique execution workflow; and *2)* the ability to break neighborhood constraint. The kernel programming API of CLIP is still "edge function", which is very similar to X-Stream and GridGraph and hence will not much affect the programmability.

4.1 Workflow

CLIP uses the same data model as X-Stream and GridGraph, where the data is modeled as a directed graph and only the property of vertices can be modified. Figure 2 illuminates the main workflow of CLIP in detail. Its procedure is split into two phases. The first phase **sorting** is a pre-processing procedure that sorts all the edges according to a specific order defined by users. We provide a simple interface to allow the assignment of the user-defined identifier for each edge. The system will sort edges according to the identifiers. With this flexible API, users can not only achieve the grid-based partition(similar to GridGraph), but also can deal with more complex sorting requirements. In our experiments, we observe that other orders (e.g., sorting by source only) may be helpful in certain cases (e.g., memory size is enough for caching all the vertices).

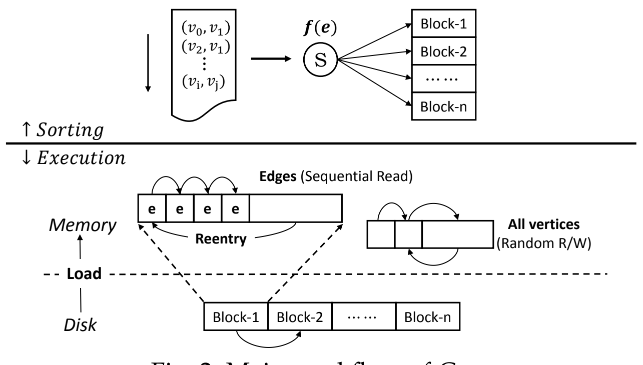


Fig. 2: Main workflow of CLIP.

The second phase **execution** is an iterative procedure that circularly reads edges until the property of vertices are converged. Within each iteration, CLIP loads and processes each of the data block by executing the user-defined "edge function" on every edge. Traditional graph processing systems restrict that each data block is processed with only one execution pass in an iteration. In CLIP, each loaded data block is processed by multiple execution passes until all the vertices/edges become inactive. Moreover, we allow users to specify a maximum reentry times (MRT), which is the maximum number of passes that will be executed for every loaded data block. MRT is useful (5-10 is usually enough) when most further local updating will be invalidated by global updating. Moreover, CLIP also uses asynchronous I/O to further improve the performance by overlapping the computation and disk I/O.

4.2 APIs

The programming interface of CLIP is defined in Table 1. This simple API is similar to those provided by existing edge-centric out-of-core systems [3], [4]. **Sort()** and **Exec()** are used to execute **one** iteration of the sorting and execution phase, respectively. To facilitate the users, we also provide a **VMap()** function that iterates every vertex and applies the user-defined input function. Table 1 also defines the type of input parameters and return value of each API function. The input parameter of user-defined function $\mathcal{F}_e$ and $\mathcal{F}_v$ both contain v_list with type $Vertices$. $Vertices$ is a container by which we can access the property of an arbitrary vertex (mmap-ed into the address space).

Specifically, the input of `Sort()` is a user-defined function $\mathcal{F}_s$ that accepts an edge as input and returns a *double* as the edge's identifier. After the sorting phase, users of CLIP may repeatedly call the function `Exec()` to perform the execution phase for updating the property of vertices. During an iteration, the user-defined function $\mathcal{F}_e$ is applied to edges (potentially multiple times) and can update the property of arbitrary vertices.

Our system also supports vertex-based and edge-based selective scheduling, which enable us to skip an edge or even a whole block if it is not needed (more detailed in Section 4.3). Specifically, through the v_list argument, $\mathcal{F}_e$ can both modify the property of an arbitrary vertex and set its activity. Moreover, $\mathcal{F}_e$ can return a *bool* to specify the state of an edge. We define that *1)* an edge is inactive if its source vertex is inactive (vertex-based) or itself is inactive (edge-based); and *2)* an entire block is inactive if all the edges it contains are inactive. CLIP automatically maintains the activity of every edge/block and uses this information to avoid the unnecessary execution.

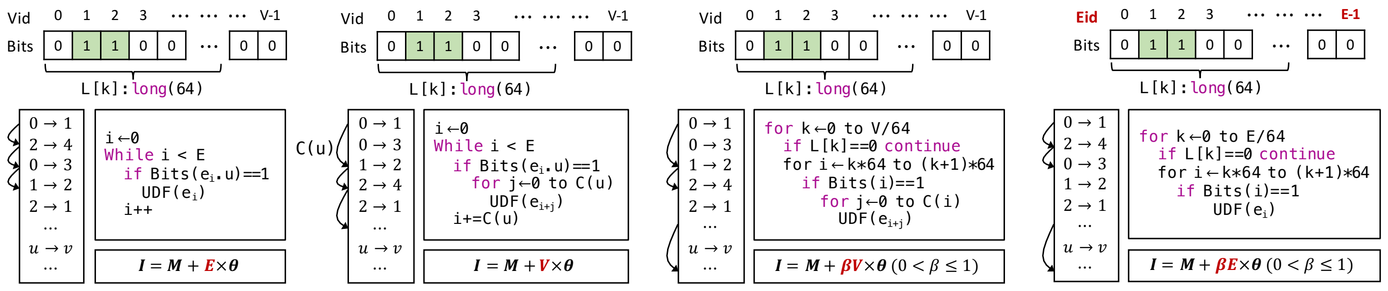
TABLE 1: Programming model of CLIP.

Sort($\mathcal{F}_s$)	—	$\mathcal{F}_s$:= **double** function($Vertices\ \&v\_list,\ Edge\ \&e$)
Exec($\mathcal{F}_e$)	—	$\mathcal{F}_e$:= **bool** function($Vertices\ \&v\_list,\ Edge\ \&e$)
VMap($\mathcal{F}_v$)	—	$\mathcal{F}_v$:= **void** function($Vertices\ \&v\_list,\ VertexID\ \&vid$)

4.3 Selective Scheduling

Selective scheduling is a very useful mechanism to avoid unnecessary CPU consumption and disk I/O. Compared to the prior systems, CLIP provides a richer and more efficient selective scheduling mechanism. Figure 3 shows in detail the execution flow and CPU consumption of the four different mechanisms. For the convenience of explanation, we use M to denote the cost of executing the user-defined function(UDF) on active edges, θ to represent the cost of executing Bits() once, and I to indicate the total cost of CPU consumption. As we can see, the mechanisms fall into two categories: 1) vertex-based selective scheduling mechanism and 2) edge-based selective scheduling mechanism. CLIP supports both selective scheduling mechanisms and can be selected at initialization. For each application, in order to achieve the correctness and efficiency, the user use either "vertex-based" or "edge-based" selective scheduling, which can't be automatically selected by CLIP.

The vertex-based selective scheduling is implemented by maintaining the current activity of vertices with a bit-array. In other words, each element of the bit array represents the state of its corresponding vertex. The vertex-based(I) (Figure 3(a)) is supported by the existing out-of-core systems (e.g., GridGraph). Although this mechanism allows the user to skip a whole block which doesn't contain any active edges, it has to check the state of each edge in the block that can't be skipped. Therefore, the CPU consumption is $\mathbf{I} = \mathbf{M} + \mathbf{E} * \theta$. According to our evaluation, this mechanism significantly affects performance when using fast storage media or running in all-in-memory mode (more detailed in Section 9). In contrast, thanks to the provided `Sort()`, CLIP supports relatively fine-grained skipping (skip edges). One can use `Sort()` to sort the input edges according to their source vertex. Therefore, edges that have the same source vertex will be placed continuously and hence can be skipped at once if the source is inactive (no need of checking the source ID for every edge, Figure 3(b)). Obviously, the CPU consumption is $\mathbf{I} = \mathbf{M} + \mathbf{V} * \theta$. Moreover, we further improve this mechanism (Figure 3(c)) by iterating vertices rather than iterating edges. Since we use a



```
i←0
While i < E
  if Bits(e_i.u)==1
    UDF(e_i)
  i++
```
$I = M + E \times \theta$

```
i←0
While i < E
  if Bits(e_i.u)==1
    for j←0 to C(u)
      UDF(e_{i+j})
  i+=C(u)
```
$I = M + V \times \theta$

```
for k←0 to V/64
  if L[k]==0 continue
  for i←k*64 to (k+1)*64
    if Bits(i)==1
      for j←0 to C(i)
        UDF(e_{i+j})
```
$I = M + \beta V \times \theta \ (0 < \beta \leq 1)$

```
for k←0 to E/64
  if L[k]==0 continue
  for i←k*64 to (k+1)*64
    if Bits(i)==1
      UDF(e_i)
```
$I = M + \beta E \times \theta \ (0 < \beta \leq 1)$

(a) Vertex-based (I) (b) Vertex-based (II) (c) Vertex-based (III) (d) Edge-based

Fig. 3: Selective Scheduling. M denotes the cost of executing user defined function(UDF) on active edges, θ represents the cost of executing Bits() once, I means the total cost of CPU consumption.

variable of type `long` to represent the state of 64 vertices, we can skip these 64 vertices if the value of this variable is 0. As a result, the CPU consumption is reduced to $\mathbf{I} = \mathbf{M} + \beta * \mathbf{V} * \theta, \mathbf{0} < \beta \leq \mathbf{1}$. In addition to the reduction in CPU consumption, this mechanism can help CLIP to only load the active edges which can further reduce the disk I/O for each iteration.

However, the vertex-based selective scheduling is not useful if the user only wants to ignore an edge in next iteration. Therefore, CLIP also provides edge-based selective scheduling (Figure 3(D)). Similar to the vertex-based method, the edge-based selective scheduling also maintains the current activity of edges with a bit-array. The difference is that it is stored on the disk. The user can specify the state of an edge by using the return value of **Exec()**. Before an edge block is loaded, CLIP will load and check its corresponding bit-array first. If there aren't any active edges, this block will be ignored. Meanwhile, this mechanism also provides fine-grained skipping in 64-edge units similar to vertex-based(III). Therefore, the CPU consumption is only $\mathbf{I} = \mathbf{M} + \beta * \mathbf{E} * \theta, \mathbf{0} < \beta \leq \mathbf{1}$

It is worth mentioning that the edge-based selective scheduling also supports user to permanently delete edges. The user can enable this function at program initialization. At the beginning of the computation, all edges will be stored in disk and CLIP executes the out-of-core processing. When the memory can hold all the remaining edges, CLIP will place these edges in memory and regenerate a new bit-array for the remaining edges. After it, CLIP will execute the in-memory processing which can significantly speed up the computation.

4.4 Sorting

External sorting is a classical problem that has been studied for many years [14], [15]. However, traditional algorithms based on merge sort is complex and require $O(logn)$ iterations. Therefore, prior out-of-core graph processing systems usually use a bucket sort based implementation. Different from prior systems that only sorting edges by vertices ID, our system allows users to define a custom comparator by setting the identifier of each edge. Without prior knowledge of the distribution of these identifiers (we even do not know the maximum/minimum value of these identifiers), it is hard to set the bucket boundary.

To solve this problem, we design and implement an algorithm that costs at most 3 iterations to sort edges and partition them into blocks that can be held in memory. In the rest of this section, we will use $\mathbf{M}$ to denote the maximum number of edges that can be held in memory; $\mathbf{N}$ to denote the total number of edges; $\mathcal{F}_s$ to denote the user-defined function that assigns an identifier to each edge. Then, the procedure of our sorting algorithm is:

Step 1: In the first iteration, CLIP reads all the edges once and determines the bucket boundaries. During this iteration, unsorted edges are first equally divided into $P = \lceil N/M \rceil$ blocks (logical division, no need to read/write disk). Each block is *1)* loaded into memory; and then *2)* sorted according to edges' identifier; finally *3)* divided into P equal divisions. The demarcations of these divisions, namely $d_0 \sim d_P$, are extracted and recorded for further usages (d_0 is the smallest identifier of an edge block; and d_P is the largest; $d_i \sim d_{i+1}$ contains about $1/P$ edges between them).

Step 2: After step 1, a total of $P * (P + 1)$ demarcations are collected and they will be used as bucket boundaries in step 2. As a result, the second iteration simply reads the edges and splits them into $P * (P + 1) - 1$ buckets.

Step 3 Finally, CLIP loads each bucket again, sorts the edges in memory and writes the result to disk. Two successive blocks are merged if their total size is less than M.

It is obvious that our algorithm is correct, as all the edges are finally sorted by their identifiers. But, in order to verify the feasibility of our algorithm, we also need to prove that at most M edges are dispatched to each bucket. This is necessary because, in Step 3, each bucket must be entirely loaded into memory for in-memory sorting. To be more explicit, in the rest of this section, we will use d_i and d_{i+1} to represent two successive demarcations in the final merged and sorted demarcation list (e.g., $d_1 = b_0$ and $d_2 = a_1$ in the above example). Then, the above conjecture can be formalized as proving: (1) $|[e|e \in edges, d_i \leq \mathcal{F}_s(e) < d_{i+1}]| < M$.

To prove (1), we can first prove that: for each of the P blocks loaded in Step 1, there are at most N/P^2 edges have identifiers within $[d_i, d_{i+1})$, i.e., (2) $|[e|e \in one\ block, d_i \leq \mathcal{F}_s(e) < d_{i+1}]| < N/P^2$. This is because that, with (2), the number of edges in each bucket is less than $N/P^2 * P = N/P$, which is less than M. Then, in order to prove (2), we can use "proof by contradiction". If there are more than N/P^2 edges in a block that has identifiers within $[d_i, d_{i+1})$, there must be another demarcation d_x that $d_i < d_x < d_{i+1}$, because two successive demarcations in the same block have only N/P^2 edges in between. However, this is a contradiction of our prior assumption that "d_i and d_{i+1} are two successive demarcations in the final merged and sorted demarcation list".

4.5 Examples

To illustrate the usages of CLIP's API, this section presents the implementation of SSSP and WCC, which benefit from loaded data reentry and beyond-neighborhood optimization, respectively.

SSSP In SSSP, a property "distance" is attached to each edge and the shortest path is defined as the lowest aggregating distance of all the edges along the path. Similar to other systems, we use a relaxing-based algorithm to solve this problem [16]. Algorithm 1 illustrates the pseudo-code of this algorithm. The `VMap` function is called in the beginning for initialization, which is followed by a series of execution iterations. Each of these iterations executes the same edge function $\mathcal{F}_e$ on every edge, which modifies the distance property of the edge's destination vertex and sets it to active. Note that this SSSP implementation is almost the same as original ones, because the trade-off between execution time and disk time is modulated only by MRT.

Algorithm 1 SSSP Algorithm in CLIP.

```
Functions:
  $\mathcal{F}_v(v\_list, vid)$ :— {
    if $vid == start$ do
      $v\_list[vid].dist \leftarrow 0$;
      $v\_list.setActive(vid, true)$;
    else $v\_list[vid].dist \leftarrow INF$;
      $v\_list.setActive(vid, false)$; }
  $\mathcal{F}_e(v\_list, e)$ :— {
    $dist \leftarrow v\_list[e.src].dist + e.weight$
    if $v\_list[e.dst].dist > dist$ do
      $v\_list[e.dst].dist \leftarrow dist$;
      $v\_list.setActive(e.dst, true)$;
    else $v\_list.setActive(e.dst, false)$; }
Computation:
  VMap($\mathcal{F}_v$);
  Until convergence:
    Exec($\mathcal{F}_e$);
```

WCC Different from the label-propagation based algorithm used by prior systems, our algorithm builds a disjoint-set over the property of vertices and uses it to solve WCC for an arbitrary graph with only one iteration. Disjoint-set, also named union-find set, is a data structure that keeps track of a set of elements partitioned into a number of disjoint subsets. It supports two useful operations: *1) find(v)*, which returns an item from v's subset that serves as this subset's representative; and *2) union(u, v)*, which joins the subsets of u and v into a single subset. Typically, one can check whether two items u and v belong to the same subset by comparing the results of *find(u)* and *find(v)*. It is guaranteed that if u and v are from the same subset then *find(u) == find(v)*. Otherwise, one can invoke a *union(u, v)* to merge these two subsets.

Algorithm 2 presents the code of our disjoint-set based WCC algorithm. Figure 4 gives an example. In our implementation, each vertex maintains a property pa that stores the ID of a vertex. If $pa[u] = v$, we name that the "parent" of vertex u is v. Vertex u is the representative of its subset if and only if $pa[u] = u$. Otherwise, if $pa[u] \neq u$, the representative of u's subset can only be found by going upstream along the pa property until finding a vertex that satisfies the above restriction (i.e. function *find* in Algorithm 2). For example, if $pa[3] = 2$, $pa[2] = 1$, $pa[1] = 1$, the subset representative of all these three vertices is 1. The *union* function is implemented by finding the representative of the two input vertices' subset and setting one's pa to another. Therefore, the whole procedure of our WCC algorithm can be simply implemented by applying the *union* function to every edge.

In Figure 4 (a), the graph has 4 vertices and 3 edges, the pa of every vertex is illustrated by arrows in Figure 4 (b). At the beginning of our algorithm, each vertex belongs to a unique disjoint subset. Hence, all arrows point to itself (1 in Figure 4(b)). During the execution, the first edge read is $(1, 2)$, so their subsets are union-ed by pointing vertex 2's arrow to 1 (2 in Figure 4(b)). In the second step, edge $(2, 3)$ is read and their subsets are also union-ed. By going toward upstream of vertex 2's arrow, we can find that its representative is 1. As a result, the union is performed by pointing vertex 3's arrow to vertex 1 (3 in Figure 4(b)). Similarly, the arrow of vertex 4 is redirected to vertex 1 after reading edge $(3, 4)$ (4 in Figure 4(b)). Eventually, all arrows point to vertex 1 and hence we found that there is only one weak connected component in the graph.

Algorithm 2 WCC Algorithm in CLIP.

```
Functions:
  $\mathcal{F}_{find}(v\_list, vid)$ :— {
    if $v\_list[vid].pa == vid$ do return $vid$;
    else return $v\_list[vid].pa =$
      $\mathcal{F}_{find}(v\_list, v\_list[vid].pa)$; }
  $\mathcal{F}_{union}(v\_list, src, dst)$ :— {
    $s \leftarrow \mathcal{F}_{find}(v\_list, src)$;
    $d \leftarrow \mathcal{F}_{find}(v\_list, dst)$;
    if $s < d$ do $v\_list[d].pa \leftarrow v\_list[s].pa$;
    else if $s > d$ do $v\_list[s].pa \leftarrow v\_list[d].pa$; }
  $\mathcal{F}_e(v\_list, e)$ :— { $\mathcal{F}_{union}(v\_list, e.src, e.dst)$; }
  $\mathcal{F}_v(v\_list, vid)$ :— {
    $v\_list[vid].pa \leftarrow vid$;
    $v\_list.setActive(vid, true)$; }
Computation:
  VMap($\mathcal{F}_v$);
  Exec($\mathcal{F}_e$);
```

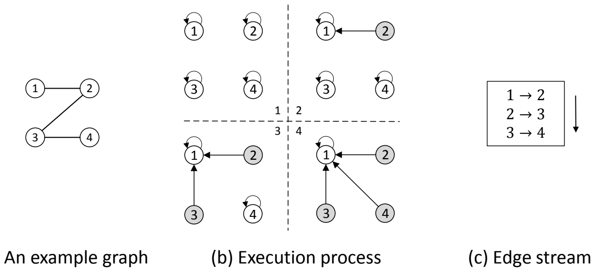


Fig. 4: WCC example.

As one can imagine, this disjoint-set based algorithm always requires only one iteration to calculate WCC for an arbitrary graph. However, when accessing the property of a vertex, it also needs to access its parent's property (breaking the neighborhood constraint). Thus, in an extreme case that the property of vertices cannot be all cached and the accesses to parents show great randomness, which may lead to very bad performance. However, this problem can be avoided by two simple optimizations: *1)* when calling *union* on two vertices, always uses the vertex that has smaller ID as the parent; and *2)* iterate the edge grids by row, which means that the grids are read in the order of "(0, 0), (0, 1), ..., (0, P-1), (1, 0), ..."(partitioned into $P \times P$ grids). According to our evaluation, these two simple optimizations can make sure that most of the parents are stored in the first several pages of vertex property and hence show good locality.

5 Semi-External Mode

As we have mentioned before, there are certain kinds of applications require the ability of randomly accessing vertices to achieve their best performance especially for the beyond-neighborhood applications. By holding all vertex data in memory, the **semi-external** mode provides a better way to deal with this requirement.

This mode is used by existing systems such as FlashGraph [17] and Graphene [18]. Many out-of-core systems, like GraphChi [2] and X-Stream [3], also support an optional semi-external mode to provide better performance. However, they can't fully play the advantages of semi-external mode due to their neighborhood constraint. In contrast, CLIP provides more expressive programming model, which can make this mode more effective. Evaluation results of CLIP reveal that CLIP achieves a drastic speedup over the other state-of-art systems that support semi-external mode, up to two orders of magnitude for certain cases.

In the rest of this section, we will *1)* present an analysis of why this mode is feasible for many important real-world use cases, *2)* describe how CLIP play the advantages of this mode more effectively by using Minimum Cost Spanning Tree (MCST) as an example, *3)* explain what efforts we do to further improve our original semi-external mode.

5.1 Semi-External Mode

Although semi-external is a natural solution for handling random vertex accesses, a natural question is: whether it is practical to put all vertices in memory for real-world graphs? Our investigation shows that, due to the sparsity of natural graphs, the answer is *YES*. Take the largest openly-available dataset Yahoo Web as an example, it contains only 1.4B vertices but 6.6B edges. Table 2 shows the actual property size for various algorithms. As we can see from this table, a typical setting of 16 GB memory is enough for holding all the vertices. According to recent market prices, an ordinary set of 16GB (2*8GB) DDR3 chips costs only $68 [19] and is affordable to most people. In contrast, the edge data is too large for an ordinary user.

TABLE 2: Data size for Yahoo graph [20].

	Size of each		Total size of	
Algorithms	Vertex	Edge	Vertices	Edges
BFS	4B	8B	5.3GB	49.4GB
SSSP	4B	12B	5.3GB	74.2GB
WCC	4B	8B	5.3GB	98.9GB
MIS	1B	8B	1.3GB	98.9GB
MCST	8B	24B	10.5GB	297GB
Coloring	5B	8B	6.58GB	98.9GB

This estimation is even valid for industry workloads. Researchers in Facebook declared in their paper "One Trillion Edges: Graph Processing at Facebook Scale" [21] that industry graphs "can be two orders of magnitude larger" than popular benchmark graphs, which means "hundreds of billions or up to one trillion edges". But, even for such huge graphs, the number of vertices is only about one billion (288M vertices and 60B edges for Twitter, 1.39B vertices and 400B edges for Facebook). $O(|V|)$ space complexity is still practical for those datasets, as even 32 GB memory is prevalent now. However, the edge data (>100GB$\sim$TB-level) is too large for memory, which can't be held in memory. In a sense, we argue that semi-external is not only practical but should be a *preferable model* given the current memory cost and graph data size.

5.2 Supporting Semi-External in CLIP

As we have mentioned above, we use mmap to directly map vertices into the virtual memory space. This mechanism makes that the semi-external mode is naturally supported by CLIP. One needs to do nothing but simply writing an application that issues random accesses to the vertices. Then, the built-in caching mechanism of mmap will automatically reserve the properties of vertices in memory if space is enough. To demonstrate this capability, in the following of this section, we will use Minimum Cost Spanning Tree (MCST) as an example.

Algorithm 3 MCST Algorithm in CLIP.

```
Functions:
  $\mathcal{F}_{find}(v\_list, vid)$ :— {
      if $v\_list[vid].pa == vid$ do return $vid$;
      else return $\mathcal{F}_{find}(v\_list, v\_list[vid].pa)$; }
  $\mathcal{F}_{union}(v\_list, src, dst)$ :— {
      $s \leftarrow \mathcal{F}_{find}(v\_list, src)$;
      $d \leftarrow \mathcal{F}_{find}(v\_list, dst)$;
      if $s < d$ do $v\_list[d].pa \leftarrow v\_list[s].pa$;
      else if $s > d$ do $v\_list[s].pa \leftarrow v\_list[d].pa$;}
  $\mathcal{F}_e(v\_list, e)$ :— { $\mathcal{F}_{union}(v\_list, e.src, e.dst)$; }
  $\mathcal{F}_v(v\_list, vid)$ :— {
      $v\_list[vid].pa \leftarrow vid$;
      $v\_list.setActive(vid, true)$; }
  $\mathcal{F}_s(v\_list, e)$ :— { return e.weight; }
Computation:
  Sort($\mathcal{F}_s$);
  VMap($\mathcal{F}_v$);
  Exec($\mathcal{F}_e$);
```

MCST Minimum Cost Spanning Tree is an important graph application that calculates a special spanning tree for a connected, undirected graph. This spanning tree connects all the vertices of the graph together with the minimal total weighting of edges. However, due to the neighborhood constraint, the existing out-of-core graph systems usually solve this problem by using GHS algorithm [22]. GHS algorithm is a parallel algorithm which starts by letting each individual node be a fragment and joining fragments in a certain way to form new fragments. This process of joining fragments repeats until there is only one fragment left and a minimum cost spanning tree is formed eventually. Although the GHS algorithm can benefit from using more threads, its performance is not comparable with the sequential Kruskal's algorithm [23] in an out-of-core environment.

In graph theory, the Kruskal's algorithm is a greedy algorithm for finding MCST of a connected weighted graph. It works sequentially by *1)* first assuming that each node constitutes a single tree in the forest; then *2)* iterating the edges from the least-weight edge to the highest-weight one); *3)* for each edge, adding it to the MCST if it connects any two trees in the forest. This algorithm is not supported by existing out-of-core graph processing systems because *1)* its implementation needs to break the neighborhood constraint; and *2)* it requires a special processing order of the edges (ascendingly ordered by their weight). These two requirements are not met by existing works, **even** those that also supports semi-external mode. In contrast, CLIP provides both the "beyond neighborhood" optimization and a sophisticated sorting method.

Algorithm 3 presents the code of Kruskal's algorithm implemented by CLIP, and Figure 5 gives an example of the algorithm's execution process. At first, the program uses `Sort()` function of CLIP to sort all the input edges according to the weight of edge and then loads the edges in an ascending order. Similar to the implementation of WCC algorithm, we also use union-find set to judge whether an edge can be added to the minimum cost spanning tree.

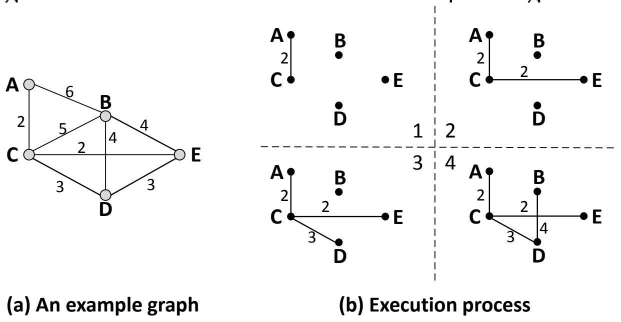


Fig. 5: MCST example.

As we can see from Figure 5 (a), the example graph has 6 vertices and 8 edges (each edge is represented as (source, target, weight)) hence these 6 vertices are initially assigned to its own tree (a tree contains only one vertex). Then, the edges are iterated by the defined order. At the beginning, edge (A,C,2) is read because it has the smallest weight. The algorithm finds vertex A and vertex C belong to different trees so that it connects them by adding edge (A,C,2) to the MCST (sub-figure 1 of Figure 5 (b)). In the following two steps, edge (C,E,2) and edge (C,D,3) are read in order and both these two edges are added to the MCST as they can also connect two trees. However, in the next step, edge (D,E,3) is read but only dropped because vertex D and vertex E already belong to the same subset (sub-figure 3 in Figure 5 (b)). This kind of steps is repeated until all the edges are read. In the following steps, only edge (B,D,4) is added to MCST and the other edges are simply dropped. As we can see, the MCST can be produced by only one iteration.

Since the vertex accessing sequence of the Kruskal's algorithm is determined by the weight of edges, it leads to completely **random access** to the vertex property. Therefore, only the semi-external mode is suitable for the Kruskal's algorithm. Based on the traditional programming model, GraphChi (and FlashGraph) failed to fully leverage the nice property of semi-external. CLIP demonstrates how to do it and its huge potential. The significant performance improvement is not due to incremental techniques but is achieved by the reduced number of iterations (and therefore the *reduced total amount of disk I/O*) that is only possible in our new programming model.

6 EVALUATION

In this section, we present our evaluation results on CLIP and compare it with the state-of-art systems X-Stream and GridGraph. We split all the benchmarks we tested into two categories by their properties and discuss the reason of our speedup respectively.

6.1 Setup

6.1.1 Environment

All our experiments are performed on a single machine that is equipped with two Intel(R) Xeon(R) CPU E5-2640 v2 @ 2.00GHz (each has 8-cores), 32GB DRAM (20MB L3 Cache), and a standard 1TB SSD. According to our evaluation, the average throughput of our SSD is about 450MB/s for sequential read. We use a server machine rather than an ordinary PC for the testing because we want to test CLIP more comprehensively including multi-threaded and different memory limits.

TABLE 3: The real-world graph datasets. A random weight is assigned for unweighted graphs.

Graph	Vertices	Edges	Type	Threshold	
				external	semi
LiveJournal [24]	4.85M	69.0M	Directed	16MB	256MB
Dimacs [25]	23.9M	58.3M	Undir.	64MB	256MB
Twitter [26]	41.7M	1.47B	Directed	128MB	4GB
Friendster [27]	65.6M	1.8B	Directed	128MB	4GB
Yahoo [20]	1.4B	6.64B	Directed	4GB	8GB

6.1.2 Benchmarks

We consider two categories of benchmarks. The first category is **asynchronous applications**, which includes SSSP, BFS and other algorithms like delta-based PageRank [28], diameter approximation [29], transitive closures [30], betweenness centrality [31], etc. For this kind of applications, the same relaxation based algorithms can be implemented with CLIP as in X-Stream and GridGraph. The only difference is that the user of CLIP can inform the system to enable loaded data reentry by setting MRT. The second category is **beyond-neighborhood applications** (e.g., WCC, MIS, Graph Stream Algorithms [11], [13]), which require users to develop new algorithms to achieve the best performance.

6.1.3 Methodology

The main performance improvement of CLIP is achieved by reducing the number of iterations with more efficient algorithms. Thus, if all the disk data is cached in memory (which is possible as we have a total of 32GB memory), we cannot observe the impact of disk I/O on overall performance. In order to demonstrate our optimizations in a realistic setting with disk I/O, we use cgroup to set various **memory limits** (from 16MB to 32GB).

Specifically, for every combination of (system, application, dataset), we test three different scenarios: *1)* **all-in-memory**, i.e., limit is set to 32GB so that most of the tested datasets can be fully contained in memory; *2)* **semi-external**, where the memory limit is enough for holding all the vertices but not all the edges; and *3)* **external**, where the memory limit is extremely small so that even vertices cannot be fully held in memory. As the number of vertices and edges are different for different datasets, the thresholds used for semi-external and external are also dataset-specific. The exact numbers are presented in Table 3, from which we can see that the limit is down to only 16MB as the vertex number of LiveJournal is less than 5M.

Moreover, for the clarity of presentation, if not specified explicitly, we always attempt all the possible number of threads and report the best performance. This means that we use at most **16** threads for testing X-Stream and GridGraph. In contrast, we test CLIP with **16** threads for asynchronous applications but only **one** thread for beyond-neighborhood algorithms.

TABLE 4: Execution time (in seconds) On SSSP/BFS. For each case, we report the results of all three scenarios in the format of “external / semi-external / all-in-memory”. ‘-’ is used if we cannot achieve all-in-memory even when the limit is set to 32GB. Since X-Stream requires extra memory for shuffling the messages, 32GB is not enough even for smaller datasets like Friendster and Twitter. ‘∞’ means that the application does not finish after running 24 hours.

		LiveJournal	Dimacs	Friendster	Twitter	Yahoo
SSSP	X-Stream	357.9 / 118.4 / 8.45	77212 / 22647 / 853.2	6352 / 3346 / -	4065 / 2255 / -	∞ / ∞ / -
	GridGraph	66.42 / 48.1 / 6.97	14618 / 13480 / 889.9	1086 / 784.6 / 85.31	1639 / 1083 / 83.51	77298 / 17432 / -
	CLIP	30.14 / 11.23 / 5.09	3202 / 1981 / 316.1	176.2 / 55.79 / 55.85	1353 / 600.6 / 91.82	18160 / 6932 / -
BFS	X-Stream	91.50 / 22.94 / 4.06	8934 / 6538 / 114.9	2526 / 1084 / -	1421 / 627.4 / -	∞ / ∞ / -
	GridGraph	13.20 / 15.4 / 2.49	5199 / 5239 / 406.2	499.6 / 493.7 / 61.54	220.5 / 209.6 / 32.16	35572 / 7403 / -
	CLIP	10.01 / 5.46 / 2.53	1768 / 1059 / 96.12	98.87 / 38.55 / 38.72	141.2 / 110.4 / 44.7	10533 / 3297 / -

6.2 Loaded Data Reentry

We use two applications, SSSP and BFS, to evaluate the effect of loaded data reentry technique. All of them can be solved by relaxation based algorithms.

6.2.1 Comparison

The results are presented in Table 4, in which all the three different scenarios are included. In this table, ‘-’ means that we cannot achieve *all-in-memory* even when the limit is set to 32GB. and ‘∞’ means that the application does not finish after running 24 hours. As we can see, CLIP can achieve a significant speedup (1.8×-14.06×) under the *semi-external* scenario. In contrast, the speedup on *external* scenario is less (only up to 6.16×). This is reasonable because, with a smaller limit, the number of edges that can be held in memory is less, therefore, the diameter of the sub-graph loaded into memory is smaller. As a result, the effect of reentry is also weaker. Moreover, even for all-in-memory settings, CLIP still outperforms the others if the diameter of the graph is large (e.g., we achieve a 2.7× speedup on Dimacs), which is because that CLIP allows the information to be propagated faster within a sub-graph and eventually makes the convergence faster.

In order to justify the above argument, we compare the number of iterations that is needed for converge on CLIP and the other systems. Results show that our loaded data reentry technique can greatly reduce this number. This improvement is especially significant for large-diameter graphs, like Dimacs, where more than 90% of the iterations can be reduced.

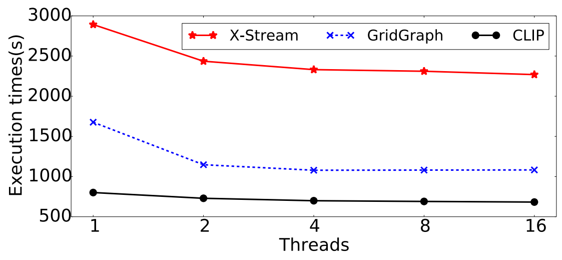


Fig. 6: The scalability for SSSP on Twitter graph, evaluated in semi-external scenario.

6.2.2 Scalability

Since we use the same algorithm as X-Stream and GridGraph, our implementation of SSSP and BFS follow the neighborhood constraint. Following neighborhood constraint makes it easy to enable the multi-thread model of CLIP to leverage the multi-core architecture. However, since disk I/O is the real bottleneck, there is actually not a big difference between using multi-thread or not.

Figure 6 illustrates our experiments results on scalability. As we can see, GridGraph has the best scalability as it can achieve a 1.55x speedup by using 4 threads. However, it is large because the single-threaded baseline of GridGraph is inefficient. In fact, the single-threaded CLIP is already faster than multi-thread version of GridGraph.

6.3 Beyond-neighborhood

6.3.1 Applications

For some problems, new algorithms need to be implemented to leverage beyond-neighborhood strategy. Besides WCC that described in Section 4.2, we introduce one more example named MIS in our evaluation.

MIS is an application that finds an **arbitrary** maximal independent set for a graph. In graph theory, a set of vertices constitutes an independent set if and only if any two of these vertices do not have an edge in between. We define that a maximal independent set as a set of vertices that *1)* constitutes an independent set; and *2)* is not a proper subset of any other independent sets. Note that there may be multiple maximal independent sets in a graphs, and MIS only requires to find one arbitrary maximal independent set from them. To solve this problem, X-Stream and GridGraph implement the same parallel algorithm that is based on Monte Carlo algorithm [32]. In contrast, we use a simple greedy algorithm to solve this problem, which consists of three steps: *1)* a *Sort()* is invoked to sort all the edges by their source IDs; *2)* a *VMap()* is called to set the property of all the vertices to *true*; and *3)* an *Exec()* is executed which iterates all the edges in order and set the property in_mis of the input edge e's source vertex to *false* if and only if “$e.dst < e.src \ \&\& \ v\_list[e.dst].in\_mis == true$”. After executing only **one** time of the *Exec()*, the final results can be obtained by extracting all the vertices whose property in_mis are *true*.

Our MIS algorithm is not only beyond-neighborhood but also requires that the edges are processed in a specific order. Thus, it is essentially a sequential algorithm that requires users to use the `Sort()` function provided by CLIP to define a specify pre-processing procedure. However, our algorithm is much faster than the parallel algorithm used by X-Stream and GridGraph, because it requires only one iteration for arbitrary graphs.

6.3.2 Comparison

Table 5 shows the evaluation results on beyond neighborhood applications. We see that CLIP can achieve a significant speed up over the existing systems on all the three scenarios: up to 2508× on *external*, up to 4264× on *semi-external*, and up to 139× on *all-in-memory*. Same as the asynchronous algorithms, the main reason of the speedup in CLIP is that

TABLE 5: Execution time (in seconds) on WCC and MIS. Format of this table is the same as Table 4. As the size of vertex property is only 1/4 of other applications in MIS, its corresponding thresholds for external and semi-external execution is also only 1/4 of the given number in Table 3, e.g., only 4MB for executing MIS with LiveJournal in external scenario.

		LiveJournal	Dimacs	Friendster	Twitter	Yahoo
WCC	X-Stream	179.5 / 57.77 / 10.25	16633 / 6751 / 185.3	4521 / 2341 / -	1904 / 1194 / -	∞ / ∞ / -
	GridGraph	22.32 / 13.8 / 3.57	6547 / 5757 / 422.5	967.5 / 466.6 / 82.95	431.5 / 272.3 / 62.3	19445 / 2916 / -
	CLIP	3.73 / 2.40 / 2.43	2.61 / 1.35 / 1.33	186 / 65.48 / 64.56	132.7 / 49.03 / 48.85	310.6 / 220.9 / -
MIS	X-Stream	422.1 / 152.6 / 13.06	103.4 / 41.42 / 5.95	9880 / 4867 / -	5513 / 3042 / -	∞ / ∞ / -
	GridGraph	166.6 / 122.1 / 2.98	46.32 / 39.19 / 14.46	3945 / 3777 / 253.7	2510 / 2473 / 156.1	∞ / ∞ / -
	CLIP	6.7 / 2.57 / 2.58	1.6 / 1.17 / 1.21	188.8 / 62.49 / 62.18	90.44 / 49.08 / 49.13	321.5 / 220.2 / -

the algorithms require much less iterations to calculate the results. The original algorithms can only converge after using tens or even thousands of iterations. In contrast, our algorithms require only one iteration for all the graphs. As a result, even if we can only use a single thread to execute our beyond-neighborhood algorithms, the large amount of disk I/O and computation avoided by this iteration reduction is enough to offer better performance than other parallel algorithms.

Moreover, as we can see from the table, even though that the algorithms used by CLIP do not follow the neighborhood constraint, they are still much faster than the other systems in the *external* scenario, where the vertices are *not* fully cached in memory. This is because that the caching mechanism of mmap is particularly suitable for processing power-law graphs. Hence, the number of pages swapping needed for vertices are moderate, at least far less from offsetting the benefit we gain from reducing redundant read of edges.

6.4 Semi-External

6.4.1 Applications

Besides MCST that we have described in Section 5.2, here we introduce another example named coloring for our evaluation of semi-external mode. In graph theory, finding the chromatic number of a graph (hereinafter coloring) means finding the smallest number of colors needed to color the vertices of a graph so that no two adjacent vertices share the same color [33]. Since this is an NP-complete problem, the current practical solutions are all approximation algorithms that try their best to find the smallest number.

However, due to the neighborhood constraint of prior systems, their implementation of coloring is typically an MIS-based algorithm that executing MIS for several times and assign a new color for the result arbitrary maximal independent set. According to our evaluation, this algorithm not only has an expensive cost but also leads to the bad result (i.e., output larger chromatic number).

In contrast, CLIP supports the Welch-Powell algorithm [34] which is a simple sequential greedy algorithm that produces the smaller chromatic number and less disk I/O. This algorithm can be implemented in CLIP with five steps: *1)* calling an *Exec()* method to calculate the out-degree of each vertex and set the property $degree$ of the corresponding vertex to this value. *2)* calling a *Sort()* method to sort all the edges by $v\_list[e.src].degree$ in descending order. *3)* calling a *VMap()* method to set each vertex's property $coloring$ to $true$ and its state to active. *4)* calling an *Exec()* to iterate all the edges in order and set the property $coloring$ of the input edge e's source vertex to *false* if and only if "$v\_list[e.dst].degree > v\_list[e.src].degree$ && $v\_list[e.dst].coloring == true$". *5)* calling a *VMap()* method to set vertex state to inactive if and only if "$v.coloring == true$". Meanwhile, the vertex will set its state to active and its property $coloring$ to $true$ if "$v.coloring == false$". Eventually, the algorithm will terminate if all vertices are in the inactive state. Otherwise, the program will jump to the fourth step and continue to execute.

Although edges with the same source vertex are contiguously stored, it's difficult to know where the outgoing edges of a vertex are located in the input graph. Obviously, it is also a completely random access to the vertex property. Therefore, the semi-external model is more suitable for this algorithm.

TABLE 6: Execution time (in seconds) for Semi-External Mode. '∞' means that the application does not finish after running 24 hours.

	LiveJournal	Dimacs	Friendster	Twitter	Yahoo
MCST					
X-Stream	75.3	26.62	2648	946.3	∞
GridGraph	230.4	5856	6654	3318	∞
CLIP	3.69	1.59	93.05	74.92	686.2
Coloring					
X-Stream	8953	83.21	∞	∞	∞
GridGraph	6250	72.49	∞	∞	∞
CLIP	32.03	5.04	1682	2273	6302

6.4.2 Comparison

Table 6 shows the evaluation results on semi-external mode. As we can see, CLIP can achieve a significant speedup over the existing systems: up to $28.46\times$ for MCST and up to $195.1\times$ for Coloring. Same as the beyond-neighborhood applications, the main reason for the speedup in CLIP is that the algorithms require much less iteration to calculate the results. For MCST application, the original algorithms will only terminate after using tens of iterations. However, our algorithm requires only one iteration for all the graphs. For Coloring, the result chromatic number is equal to the number of iterations. As we mentioned above, our algorithm produces the fewer chromatic number and hence needs less iteration, which means it is better in both performance and result. Meanwhile, we only have to iterate all the edges once for each coloring iteration, while the original MIS-based algorithms' each coloring iteration is an MIS-finding procedure that requires multiple sub-iterations. Obviously, the amount of disk I/O and computation of CLIP are significantly less than the prior systems. As a result, CLIP offer better performance even we only use a single thread.

It is worth mentioning that the performance of GridGraph is not better than X-Stream for MCST application, even if they use the same parallel algorithm. The reason is that GridGraph does not allow users to dynamically delete edges. Moreover, the GHS algorithm needs to propagate the minimum cost to its connected component for each iteration. GridGraph has to iterate all the edges. However, X-Stream only needs to iterate the edges whose state are unknown. Meanwhile, the number of these edges will be less and less.

TABLE 7: Chromatic number. Some of the exact numbers are not presented, as they do not terminate after running 24 hours.

	LiveJournal	Dimacs	Friendster	Twitter	Yahoo
X-Stream	334	5	156	1101	-
GridGraph	334	5	156	1101	-
CLIP	325	5	130	1082	984

Coloring Result Table 7 presents the result approximate chromatic number of executing coloring algorithm with each system. Since X-Stream and GridGraph cannot finish their computation within 24 hours on many large graphs (Friendster, Twitter, Yahoo), in order to get the result, we use a machine with 1TB DRAM to execute them in a full-memory mode. We can see that the algorithm implemented by CLIP can usually achieve the smaller chromatic number. In other words, CLIP can get more accurate result. It is worth mentioning that we can still achieve significant speedup even if the chromatic number required is not significantly reduced. As we mentioned in Section 6.4.2, the reason is that we only have to iterate all the edges once for each coloring while the original algorithms require multiple sub-iterations as each of their coloring iterations is an MIS-finding procedure. For example, the number of sub-iterations on Livejournal is 2744 for X-Stream and GridGraph. In contrast, the number of sub-iteration is only 325 (equal to the chromatic number) for CLIP. Moreover, due to our edge-based selective scheduling mentioned in section 4.3, all remaining edges will be cached in memory after several iterations. Therefore, the execution time of each iteration will become less and less as the number of iterations increases.

7 Discussion

7.1 Compared with In-memory System

Since the memory size of the single machine is getting bigger and bigger, most of the input graph can be held in memory [35], why can't we use in-memory system [35], [36], [37] instead of out-of-core system? In fact, compared with in-memory system, CLIP has the advantage of **low cost**, **good performance** and **better scalability**.

TABLE 8: Execution time (in seconds) for CLIP and Galois. '-' designates out of memory.

	systems	LiveJournal	Dimacs	Friendster	Twitter	Yahoo
WCC	Galois	2.58	1.81	49.75	42.36	-
	CLIP	2.4	1.35	65.48	49.03	220.9
MIS	Galois	2.01	1.36	40.14	34.15	-
	CLIP	2.57	1.17	62.49	49.08	220.2

The first advantage of CLIP is **low cost**. According to recent market prices [19], the price of 2 Terabytes of memory is $16512(requires 128 16GB-DDR3 DRAM, each $129). In contrast, the price of 2 Terabytes of HDD is only $59.99 and SATA SSD is only $472.16. Besides, a common PC can be equipped with a standard 2 Terabytes of HDD or SSD. However, 2 Terabytes of memory require a dedicated server.

Second, CLIP has a **good performance**. Table 8 presents the comparison between CLIP (semi-external mode) and Galois [36] (a state-of-art in-memory graph processing system). Since the time of loading data dominates the execution time, the performance of CLIP is indeed comparable to Galois. CLIP is slower than Galois on large datasets (Friendster, Twitter) because we use different encoding formats for the binary graph file on disk. Take "Twitter" as an example, the input edges size of WCC is 11.25GB for Galois but 21.88GB for CLIP. In summary, it is very worthwhile to get the maximum performance with the least cost.

Finally, CLIP has a better **scalability**. The memory size provided by a single machine is limited. As we mentioned above, the Facebook graph benchmark [21] has about 400B edges. This huge graph requires tens of Terabytes of memory, which is a big challenge for single machine. In contrast, the out-of-core systems can easily deal with such challenge. For example, 32 Gigabytes of memory and 48 Terabytes of disk is a prevalent configuration for a server.

7.2 Preprocessing Time

Since our algorithms of MIS, MCST and Coloring require that the input edges should be sorted in a specific order (by source ID for MIS, by weight for MCST and by degree for Coloring), we evaluate the cost of this preprocessing procedure and compare it to GridGraph (which is not included in the time reported in Table 5). Table 9 illustrates our evaluation results, which shows that we have almost the same preprocessing time as GridGraph. This is because that, even we do not have any prior knowledge of the user-defined sorting metrics, our novel external sorting algorithm can finish by reading edges for only three times. What needs further explanation is that GridGraph and CLIP (by source and by weight) are evaluated on unweighted graph and CLIP (by weight) are evaluated on the weighted graph, so their preprocessing time is different. Meanwhile, sorting by degree needs to calculate the out-degree of the vertices first, therefore it requires an extra iteration.

TABLE 9: Preprocessing time (in seconds).

	GridGraph	CLIP (By source)	CLIP (By weight)	CLIP (By degree)
LiveJournal	4.57	5.06	11.42	8.04
Dimacs	4.09	5.12	6.81	6.53
Friendster	185.5	145.3	302.2	264.3
Twitter	160.3	126.2	241.1	205.5
Yahoo	1616	1410	2111	1502

Moreover, although sometimes the preprocessing time is longer than the execution time, the total execution time is **still** less. For example, the total execution time (preprocessing+execution) of computing MIS on Friendster is 4867s for X-Stream and 3962.5s for GridGraph. In contrast, the total execution time of CLIP only is 145.3+62.49=207.79s, which is 19.07x less than GridGraph and 23.42x less than X-Stream, not to mention that the preprocessing cost can be amortized by reusing the sorting results.

8 More Optimizations

As we have mentioned before, CLIP provides a more flexible programming model than existing out-of-core graph processing systems. In former sections, we describe the capability of this expressiveness by presenting two general optimization techniques that can benefit many applications with the same data accessing pattern. In this section, we further demonstrate the possibility of this expressiveness by designing and evaluating some more optimizations.

8.1 Diagonal-first Partitioning

According to the evaluation results in Section 6.2, loaded data reentry leads to a huge reduction in the number of iterations and hence reduces the execution time. To further increase the possible speedup, in this section, we explore the impact of different graph partition strategies. Moreover, we propose a series of optimization strategies to further improve the performance of loaded data reentry both on semi-external and external mode.

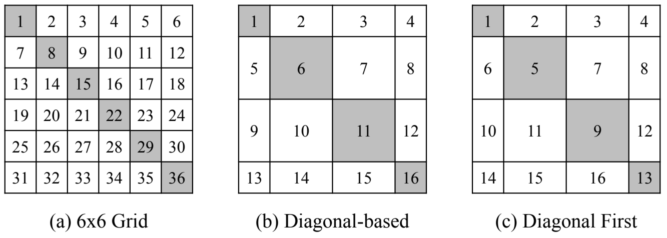


Fig. 7: Partition and iterating order.

In order to analyze which data blocks affect performance, we partition graph as grid format. Figure 7(a) illustrates a 6×6 grid partition. We find that only the diagonal blocks (gray block) need to be processed multiple times, which really affects the number of iterations. The reason is that, for consideration of I/O locality, CLIP only loads the corresponding source and target intervals of each loaded data block (each grid) into memory. Then it processes each edge of this data block in memory and updates the value of target vertex by using the value of source vertex and the weight of edge if the state of source vertex is active. Therefore, a loaded data block will be processed multiple times if and only if the value of its source vertices is updated. And the data blocks on the diagonal just meet this requirement.

Meanwhile, theoretically, the greater the number of edges in these diagonal blocks, the more vertices will be updated during the reentry, which will lead to more vertices to get the final value as soon as possible. In other words, a higher proportion of edges in the diagonal data block will reduce the number of iterations more. However, the previously used naive regular grid partition usually leads to small proportion diagonal edges, which limit the effect of the reentry optimization.

Therefore, we propose a diagonal-based partitioning strategy that tries to enlarge the proportion of diagonal edges as much as possible. Specifically, it contains three steps: 1) partitioning the graph into many blocks as the grid format and counting the number of edges of each block; 2) Using a greedy strategy that merges the small blocks near the diagonal into big ones (under the memory limit); 3) Partitioning the graph as the new partition scheme. Figure 7(b) is the result of the merger. It's clear that, after merging, more edges are included in the diagonal blocks.

Table 10 illustrates the comparison between the regular grid partitioning strategy (GridGraph) and diagonal-based partitioning strategy (CLIP-OPT) on preprocessing Twitter graph. *Threshold* represents the limit of memory size. P indicates the maximum number of partitions to ensure that each data block can be held into memory. *Proportion* means the proportion of diagonal edges. *Time* shows the preprocessing time for each strategy. As we can see, the diagonal-based partitioning strategy can achieve a bigger proportion (2.6×-3.17×). Meanwhile, the preprocessing time of diagonal-based partitioning is very close to or even shorter than the regular grid partitioning.

TABLE 10: The comparison between grid partition and diagonal-based partition on preprocessing Twitter graph.

Threshold	System	P	Proportion	Time(s)
64MB	GridGraph	128	0.04	84.84
	CLIP-OPT	20	0.11	72.93
256MB	GridGraph	64	0.06	77.37
	CLIP-OPT	9	0.19	72.77
512MB	GridGraph	32	0.1	77.1
	CLIP-OPT	6	0.26	72.31

8.2 Diagonal-first Scheduling

After the input graph is divided into grids, what needs to be determined next is the scheduling order of these data blocks. GridGraph supports two simple scheduling orders, in the order of the rows and in the order of the columns. Figure 7 shows an example of scheduling by rows. In fact, the scheduling order does not have any effect on the synchronous execution, but it will affect the convergence speed of asynchronous execution. Although GridGraph also provides asynchronous execution, its simple scheduling order does not fully exploit the efficiency of asynchronous execution.

According to our analysis, we find that the scheduling order of diagonal blocks is the key to affect the number of iterations. For example, after block 6 in Figure 7(b) is processed, the value of its source vertices will be updated. Therefore, the edges whose source vertex belong to this interval need to be processed next. Clearly, these edges are in the same row of block 6 (block 5, 7, and 8). However, if a naive "iterate by row order" is used, block 5 can only be influenced by the updates in the next iteration as block 6 is processed after it. Therefore, this order does not further reduce the number of iterations.

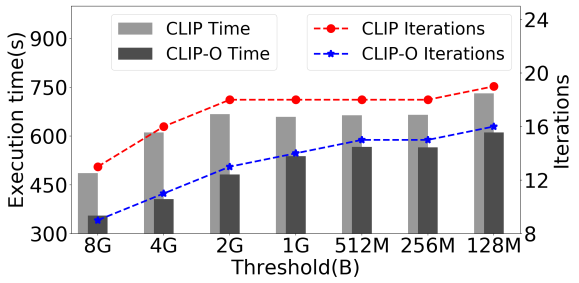


Fig. 8: Iterations and Execution Time For SSSP on Twitter.

To alleviate this problem, we propose a diagonal-first scheduling order that all the diagonal blocks are scheduled first before any other blocks in the same row. Figure 7(c) is the result order, where the number in each block represents the order. Figure 8 illustrates the number of iterations and execution time for SSSP on Twitter graph under different partitioning and scheduling strategies. CLIP-O represents the optimization version of CLIP, which uses the diagonal-based partitioning and scheduling strategy. CLIP means our original simple partitioning and scheduling strategy. As we can see, this optimization strategy can achieve a 1.17×-1.5× speedup compared with our previous work, which can be reflected by the reduction in the number of iterations. This performance improvement is mainly due to the fact that our optimization strategy can spread the updated information faster, thus speeding up the convergence of the algorithm.

9 To Meet Fast Storage Media

Since the slow storage media (HDD, SATA SSD) is the major performance bottleneck of out-of-core graph processing systems, it is worthwhile to sacrifice parallelization as long as we can reduce the disk I/O. This fact leads to our claim of using a single-threaded or an unoptimized in-memory execution manner is enough to be better than the existing out-of-core graph processing systems.

However, thanks to CLIP reduces disk I/O by adding more computations, the proportion of computation cost and I/O cost of CLIP is much more balanced than the existed graph processing systems. This result opens the opportunity of further increase CLIP's performance by extending it to a more efficient parallel system.

Even more, we further consider a new environment of faster storage media (NVMe SSD). Our experiment results demonstrate that, if NVMe SSD (2.88GB/s for sequential read) is used, the CPU consumption of CLIP can become the new bottleneck. Take executing WCC on NVMe SSD as an example, for the Twitter graph (with size 21.88GB), our single-threaded algorithm requires 23.37s to complete while the IO time is only 8.81s. In fact, this phenomenon is also widely existing in other out-of-core graph processing systems (e.g., GraphChi).

In this section, to meet this challenge, we further improve our in-memory implementation by *1)* providing more efficiently in-memory execution manner for load data reentry and *2)* supporting parallelization for the single-threaded algorithms which further perfects CLIP to make it a more efficient **parallel** out-of-core graph processing system.

9.1 Improvement

9.1.1 In-memory Optimization for Loaded Data Reentry

As we can see from Table 4, the execution time of CLIP on all-in-memory mode is close to the existing systems. However, compared with Galois, CLIP is much slower for asynchronous applications, especially on a large diameter graph. Take Dimacs Graph as an example, although it has only 58.3 million edges, the execution of BFS/SSSP are up to 96.12s/316.1s respectively. In order to maximize the performance of all-in-memory mode, we propose two optimization techniques for asynchronous applications on in-memory mode, which are used to increase the iterating speed over the edges and reduce the memory usage.

Fast Iterating In order to analyze this problem, we count the proportion of active edges in each iteration for BFS on Dimacs graph. Figure 9 illustrates our evaluation results. As we can see that the largest proportion is only about 0.04% and the average proportion is less than 0.02%. Unfortunately, existing edge-centric out-of-core systems (e.g., X-Stream, GridGraph) don't know the location of a vertex's outgoing edges. Therefore, for each iteration, they need to iterate all edges and determine whether an edge can be executed via the state of its source vertex. Although GridGraph supports a block-level selective scheduling, it needs to iterate the whole block even if it contains only one active vertex. This means that they spent most of the time iterating the useless edges.

However, by using the more efficient selective scheduling mechanism vertex-based(III) (Figure 3(c)) we propose (more detail in Section 4.3), CLIP can solve this problem easily. For each iteration, we first iterate our bit-array and then only iterate the active edges. This optimization strategy can greatly reduce the execution time, especially for the large diameter graph. Moreover, as the number of iterations increases, the number of active vertices will become less and less. CLIP will switch to using **worklist** (a container) to store the ID of all active vertices(the proportion of active edges is less than 5%), which completely avoids the CPU consumption for inactive edges.

Besides, our original semi-external mode distribute data blocks to different threads in units of 24MB. This static setting will lead to serious imbalances among different threads as the number of iterations increases. Thanks to our vertex-based(III) selective scheduling mechanism, we can clearly know the distribution of active edges. Therefore, we can implement dynamic block size settings to balance computation.

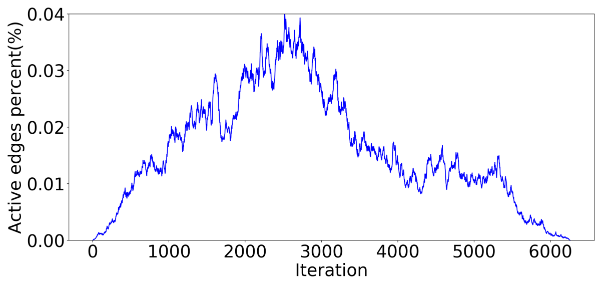


Fig. 9: The proportion of the number of active edges in each iteration for BFS on Dimacs graph.

Data Compression It is worth mentioning that CLIP can organize outgoing edges in the Compressed Sparse Row (CSR) format in memory while loading data. Although this strategy does not speed up the iterating speed, it can greatly reduce the use of memory. Take Yahoo graph as an example, its edge size can be compressed from 49.4GB to 30.09GB, which reduces the amount of memory use by 40%.

9.1.2 Parallelization for Beyond-neighborhood

As described above, our single-threaded implementation for beyond-neighborhood applications will become the new bottleneck when using faster storage media. In order to solve this problem, we new-design the parallel algorithms for beyond-neighborhood applications by using the flexible programming model of CLIP. Parallelization is an effective manner to accelerate in-memory computing performance by using multiple cores. There have been a lot of works trying to convert efficient single-threaded graph algorithms into parallel algorithms [36], [38], such as parallel disjoint-set, parallel Kruskal' algorithm and so on. The main principle of these conversions is to try their best to process the edges that have no conflicts with each other in parallel. However, these conversions are mainly for in-memory execution which leads to random access to the edges. Therefore, the parallel algorithms can't be directly applied to our out-of-core scenario (need to access the edges sequentially).

Fortunately, due to the flexible programming model of CLIP, one can easily new-design the parallel algorithms for beyond-neighborhood applications by limiting the scope of parallelism to the loaded data block (CLIP has helped the user to complete this limitation). According to our evaluation, although this limitation will reduce the performance of parallel, the parallel algorithms can still achieve significant speedup compared to their single-threaded version(more detailed evaluation in Section 9.2.2). Figure 10 shows the principle of the parallelization for the beyond-neighborhood applications. For our original single-threaded version, CLIP processes the graph with the granularity of the edges. However, we can process the graph with the granularity of the loaded data blocks. In this way, we can perform parallel algorithms on the loaded data in memory. Meanwhile, we also guarantee the original processing order of the graph. Therefore, the parallel execution does not increase the number of iterations.

According to our investigation, these parallel algorithms in memory require that *1)* the program can access the property of an arbitrary vertex and *2)* the loaded data block needs to be processed multiple times to ensure correctness [36]. Obviously, the flexible programming model of CLIP just to meet this two requirements. In order to more clearly illustrate, we use the WCC and MIS algorithms as examples to guide users how to convert these single-threaded algorithms into their parallel version.

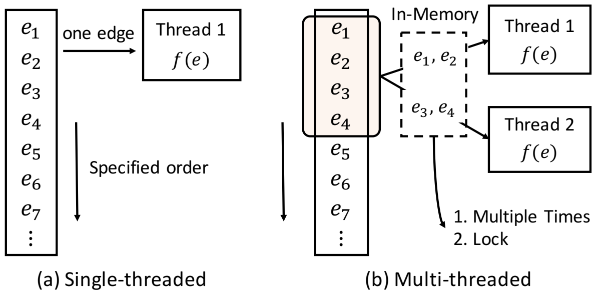


Fig. 10: The Principle of New-designed Parallel Algorithms.

Parallel WCC Algorithm 4 illustrates the pseudo-code of the parallel WCC algorithm in CLIP. Different from the single-threaded algorithm (Algorithm 2), we simply use compare-and-swap(CAS) to avoid updating the property of a vertex at the same time. In this way, subsets that do not conflict with each other are first merged in parallel. It is worth noting that, with the formation of the larger subset, the available parallelism decreases. However, because these algorithms require only very few iterations (even only one iteration), it is worthwhile to use less disk I/O (only one iteration) in exchange for slightly worse parallel performance for the semi-external mode.

Parallel MIS The parallelization of MIS also follows the principle of parallel processing the edges that have no conflicts. Unlike parallel WCC, the conflict is mainly due to the fact that the greedy algorithm requires the vertices with the smaller ID to determine their state first. In other words, a vertex will be processed as long as all the neighbors of a vertex (ID is smaller than its own) have already determined their state. However, since all edges have been sorted by their source ID, we can easily guarantee that the data blocks with smaller IDs will be processed first. For loaded data block, the parallelism can be achieved in three simple ways: 1) Processing all the outgoing edges of a vertex in parallel; 2) If the state of the neighbors of a vertex(ID is smaller than its own) have already determined their state, the outgoing edges of this vertex can begin to be processed immediately; 3) The loaded data block needs to be processed multiple times until the state of all the source vertices in this block will be determined. Fortunately, the selective scheduling mechanism and loaded data reentry exactly meet this requirement.

Algorithm 4 Parallel WCC Algorithm in CLIP.

```
Functions:
  $\mathcal{F}_{find}(v\_list, vid)$ :— {
      if $v\_list[vid].pa == vid$ do return $vid$;
      else return $v\_list[vid].pa =$
          $\mathcal{F}_{find}(v\_list, v\_list[vid].pa)$; }
  $\mathcal{F}_{union}(v\_list, src, dst)$ :— {
      while Ture do
          $s \leftarrow \mathcal{F}_{find}(v\_list, src)$;
          $d \leftarrow \mathcal{F}_{find}(v\_list, dst)$;
          if $s == d$ do break;
          if $s > d$ do $swap(s,d)$;
          if $CAS(v\_list[d].pa, d, s)$ == True do break;
  $\mathcal{F}_e(v\_list, e)$ :— { $\mathcal{F}_{union}(v\_list, e.src, e.dst)$; }
  $\mathcal{F}_v(v\_list, vid)$ :— {
      $v\_list[vid].pa \leftarrow vid$;
      $v\_list.setActive(vid, true)$; }
Computation:
  $VMap(\mathcal{F}_v)$;
  $Exec(\mathcal{F}_e)$;
```

It is worth mentioning that, similar to parallel WCC and MIS algorithms, most of the single-threaded graph algorithms can be parallelized by using the flexible programming model of CLIP, and so on. Moreover, according to our evaluation, the performance of these restricted parallel algorithms implemented in CLIP is indeed comparable to the in-memory systems (not include the loading time), such as Galois [36].

9.2 Evaluation in Memory Mode

In order to demonstrate the improvement of memory performance, we use the machine which is described in Section 6.1.1. It is worth noting that we don't consider the time of loading data into memory.

TABLE 11: Execution time (in seconds) On BFS in semi-external mode, not include the reading time. '-' designates out of memory.

	LiveJournal	Dimacs	Friendster	Twitter
X-Stream	3.95	113.7	-	-
GridGraph	1.61	405.3	23.43	10.58
CLIP	1.72	95.2	20.12	13.23
CLIP-OPT	0.31	2.19	6.54	6.11

9.2.1 In-memory Optimization for Loaded Data Reentry

Table 11 presents the comparison on semi-external mode (CLIP-OPT is the optimized version). In order not to lose justice, the execution time does not include the loading data time. As we can see, the performance of CLIP is indeed comparable to the existing out-of-core systems. However, thanks to the optimization of in-memory execution manner, CLIP-OPT achieve a significant speedup (up to 43.4×) on large diameter compared to our original system. Even for small diameter graph, CLIP-OPT can still achieve 2.21×-5.19× speedup. The improvement of the performance mainly comes from the fact that CLIP try to avoid checking the state of the useless edges.

9.2.2 *Parallelization for Beyond-neighborhood*

As mentioned in Section 9.1.2, it's very easy to convert the sequential algorithms into parallel implementation by using the flexible programming model of CLIP. In this Section, we evaluate the scalability of our new-designed parallel WCC and MIS algorithms (The parallelization of MCST and Coloring is similar to WCC and MIS respectively).

Figure 11 illustrates our evaluation results on scalability for WCC and MIS on Twitter graph. As we can see, when using 16 threads, the parallel WCC and MIS algorithms can achieve significant speedup (8.49× for WCC and 8.91× for MIS) compared to our single-threaded version.

The reason for the significant improvement in performance is that most of the real-world graph satisfy power-law distribution. Therefore, processing the outgoing edges of a vertex in parallel is very effective. Moreover, the distribution of the edges in the input graph is relatively random, so most of the merges do not create conflicts. As we use the similar parallel algorithms with Galois [36], our in-memory execution performance is even faster than some all-in-memory graph processing systems. Hence, the improved semi-external mode can meet the requirement of faster storage media or even all-in-memory scenario.

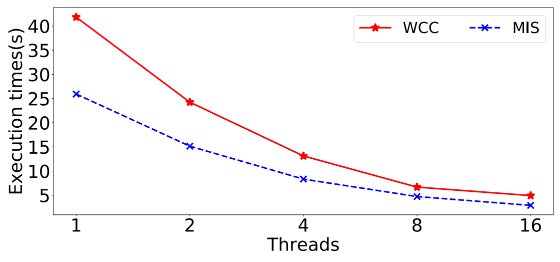


Fig. 11: The scalability for WCC and MIS on Twitter graph, evaluated in semi-external scenario, not include the reading time.

9.3 Evaluation on NVMe SSD

As we mentioned above, CLIP reduces the execution time by reducing the number of iterations and providing faster in-memory execution manner. Therefore, the performance of CLIP will be further promoted on faster storage media. In this section, we present evaluation on CLIP, the optimized version CLIP-OPT and MOSAIC [8] on NVMe SSD.

MOSAIC is a trillion-scale single heterogeneous machine out-of-core graph processing system for fast storage media (e.g., NVMe SSD) and massively parallel coprocessors (e.g., Xeon Phi) [8]. According to our evaluation, thanks to our in-memory optimization strategy, CLIP-OPT can also achieve further performance speedup (up to 2.88×) compared with CLIP. Moreover, CLIP-OPT can also achieve a significant speedup compared with MOSAIC (up to 8.01× for BFS and up to 4364× for WCC).

9.3.1 *Experiment Setup*

Due to the limitations of the experimental conditions, the experiments are performed on a common PC that is equipped with one Intel(R) Core(TM) i7-6700K CPU @ 4.00GHz (8-cores), 32GB DRAM (8MB L3 Cache) and a standard 256GB NVMe SSD. According to our evaluation, the average throughput of our NVMe SSD is about 2.88GB/s for sequential read and 1.42GB/s for sequential write. Although our machine does not equip the Xeon Phi, MOSAIC declares that it supports such a running environment. Because of the limited disk space, we only evaluate MOSAIC and CLIP on two datasets (Dimacs and Twitter). Moreover, we use BFS and WCC as representatives of asynchronous applications and beyond-neighborhood applications.

9.3.2 *Comparison*

The results are presented in Table 12, in which only the semi-external scenario is included. We see that CLIP can achieve a significant speedup over MOSAIC on WCC (up to 1514×) and BFS (up to 7.21×). Meanwhile, CLIP-OPT can further increase the speedup compared with MOSAIC (up to 8.01× for BFS and up to 4364× for WCC). Same as the previous explanation, the main reason for the speedup in CLIP is that the algorithms used by CLIP require much fewer iterations to calculate the results. Take WCC as an example, MOSAIC use the same algorithm similar to Grid-Graph which can only converge after using tens or even thousands of iterations. Although the execution time of each iteration of MOSAIC has become shorter, the huge number of iterations leads to its long execution time. However, as we can see, since the computing time has exceeded the loading time, CPU consumption of CLIP becomes the major performance bottleneck. As we mentioned above, CLIP-OPT focuses on further improving the in-memory computation performance and achieves a significant speedup, so CLIP-OPT once again pushes the performance bottleneck to the disk I/O. Evaluation result in Table 12 shows that CLIP-OPT is much faster than CLIP (up to 2.88× speedup for WCC and up to 1.11× speedup for BFS). The reason why the BFS algorithm is not obviously improved is that our previous implementation is already very close to the loading time. Obviously, if the storage media becomes faster, the speedup will be larger.

TABLE 12: Execution time(in seconds) on NVMe SSD.

Algorithms	System	Dimacs		Twitter	
		Size(GB)	Time(s)	Size(GB)	Time(s)
BFS	MOSAIC	0.41	713	8	34.41
	CLIP	0.43	98.87	10.9	19.82
	CLIP-OPT	0.43	88.97	10.9	18.89
WCC	MOSAIC	0.41	742	15	74.07
	CLIP	0.43	0.49	21.88	23.37
	CLIP-OPT	0.43	0.17	21.88	8.85

9.3.3 *Discussion*

It is worth mentioning that MOSAIC is actually an orthogonal optimization with our efforts of reducing the number of iterations. The factors that influence the execution time of out-of-core graph processing system is

$$Time \propto MAX(Load, Proc) \times Iters \quad (1)$$

"Load" is the time that loading edges into memory for each iteration. "Proc" means the execution time in memory for this iteration. "Iters" represents the number of iterations. MOSAIC mainly optimizes the two aspects of "Load" and "Proc". **1) Load:** it provides an effective data compression format which can reduce the loading time of the edge data. This optimization is at the cost of increasing the preprocessing time. As we can see from Table 12, the amount of graph size in MOSAIC is less than CLIP. **2) Proc:** using coprocessor to speed up the execution time in memory. The aim of these optimizations is to reduce the execution time of each iteration. Moreover, the performance improvements caused by these optimizations will be more significant as the disk bandwidth increases.

Different from MOSAIC, CLIP focuses on reducing the number of iterations (**Iters**). At the same time, we also make the effort to further reduce the execution time in memory (**Proc**). We believe that if it can combine with the optimization methods proposed by MOSAIC and CLIP, the performance of the single machine out-of-core graph processing system will be further improved.

10 Related Work

There are also many distributed graph processing systems. Pregel [39] is the earliest distributed graph processing system that proposes a vertex-centric programming model, which is later inherited by many other graph processing systems [3], [7]. Some existing works [40], [41], such as Giraph++ [30], have suggested to replace "think as vertex" with "think as sub-grapg/partition/embedding". They can take advantage of the fact that each machine contains a subset of data rather than only one vertex/edge and hence are much faster than prior works. However, none of these existing works could support the beyond-neighborhood algorithms used by CLIP.

Similarly, in addition to GraphChi, X-Stream and GridGraph, there are other out-of-core graph processing systems using alternative approaches [6], [17], [42], [43]. However, most of them only focus on maximizing the locality of disk I/O and still use neighborhood-constraint programming model. As a counter example, MMap [6] leverages the memory mapping capability found on operating systems by mapping edge and vertex data files in memory, which inspires the design of CLIP. But, MMap only demonstrates that mmap's caching mechanism is naturally suitable for processing power-law graphs. It does not consider the limitations of the original out-of-core systems' restrictions , which is the key contribution of this work.

There are some works [17], [18], [44] that aim to load only necessary data in an iteration, which can also reduce disk I/O. Besides, the works [45], [46], [47], [48] focus on using GPU to speed up in-memory execution. However, these methods are actually an orthogonal optimization with our efforts of reducing the number of iterations. According to our evaluation, our simple selective scheduling and in-memory optimization strategies are enough for our case.

11 Conclusion

In this paper, we propose CLIP, a novel out-of-core graph processing system designed with the principle of "squeezing out all the value of loaded data". With the more expressive programming model and more flexible execution, CLIP enables more efficient algorithms that require much less amount of total disk I/O. Our experiment results show that CLIP is up to tens or sometimes even thousands times faster than existing works X-Stream and GridGraph.

Acknowledgement This work is supported by National Key Research & Development Program of China (2016YFB1000504), Natural Science Foundation of China (61433008, 61373145, 61572280, 61133004, 61502019, U1435216), National Basic Research (973) Program of China (2014CB340402), Intel Labs China (Funding No.20160520). This work is also supported by NSF CRII-1657333, NSF SHF-1717754, NSF CSR-1717984, Spanish Gov. & European ERDF under TIN2010-21291-C02-01 and Consolider CSD2007-00050. Contact: Yongwei Wu (wuyw@tsinghua.edu.cn) and Kang Chen (chenkang@tsinghua.edu.cn).

References

[1] Z. Ai, M. Zhang, Y. Wu, X. Qian, K. Chen, and W. Zheng, "Squeezing out all the value of loaded data: An out-of-core graph processing system with reduced disk i/o," in *2017 USENIX Annual Technical Conference (USENIX ATC17)*. USENIX Association, 2017.

[2] A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi: large-scale graph computation on just a PC," in *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012, pp. 31–46.

[3] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-Stream: edge-centric graph processing using streaming partitions," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 472–488.

[4] X. Zhu, W. Han, and W. Chen, "GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, 2015, pp. 375–386.

[5] P. Yuan, C. Xie, L. Liu, and H. Jin, "Pathgraph: a path centric graph processing system," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 10, pp. 2998–3012, 2016.

[6] Z. Lin, M. Kahng, K. M. Sabrin, D. H. P. Chau, H. Lee, and U. Kang, "Mmap: Fast billion-scale graph computation on a pc via memory mapping," in *Big Data (Big Data), 2014 IEEE International Conference on*. IEEE, 2014, pp. 159–164.

[7] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012, pp. 17–30.

[8] S. Maass, C. Min, S. Kashyap, W. Kang, M. Kumar, and T. Kim, "Mosaic: Processing a Trillion-Edge Graph on a Single Machine," in *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*, Belgrade, RS, Apr. 2017.

[9] H. N. Gabow and R. E. Tarjan, "A linear-time algorithm for a special case of disjoint set union," *Journal of computer and system sciences*, vol. 30, no. 2, pp. 209–221, 1985.

[10] R. E. Tarjan and J. Van Leeuwen, "Worst-case analysis of set union algorithms," *Journal of the ACM (JACM)*, vol. 31, no. 2, pp. 245–281, 1984.

[11] A. McGregor, "Graph stream algorithms: a survey," *ACM SIGMOD Record*, vol. 43, no. 1, pp. 9–20, 2014.

[12] S. Muthukrishnan, *Data streams: Algorithms and applications*. Now Publishers Inc, 2005.

[13] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang, "On graph problems in a semi-streaming model," in *International Colloquium on Automata, Languages, and Programming*. Springer, 2004, pp. 531–543.

[14] D. E. Knuth, *The art of computer programming: sorting and searching*. Pearson Education, 1998, vol. 3.

[15] J. S. Vitter, "Algorithms and data structures for external memory," *Foundations and Trends® in Theoretical Computer Science*, vol. 2, no. 4, pp. 305–474, 2008.

[16] R. Bellman, "On a routing problem," *Quarterly of applied mathematics*, pp. 87–90, 1958.

[17] D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay, "FlashGraph: Processing billion-node graphs on an array of commodity SSDs," in *13th USENIX Conference on File and Storage Technologies (FAST 15)*, 2015, pp. 45–58.

[18] H. Liu and H. H. Huang, "Graphene: Fine-grained io management for graph computing." in *FAST*, 2017, pp. 285–300.

[19] "https://www.amazon.com."

[20] "G2 - Yahoo! AltaVista Web Page Hyperlink Connectivity Graph, circa 2002. http://webscope. sandbox.yahoo.com/."

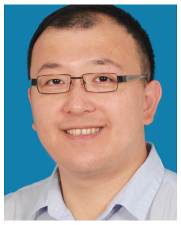
[21] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, "One trillion edges: graph processing at Facebook-scale," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1804–1815, 2015.

[22] R. G. Gallager, P. A. Humblet, and P. M. Spira, "A distributed algorithm for minimum-weight spanning trees," *ACM Transactions on Programming Languages and systems (TOPLAS)*, vol. 5, no. 1, pp. 66–77, 1983.
[23] J. B. Kruskal, "On the shortest spanning subtree of a graph and the traveling salesman problem," *Proceedings of the American Mathematical society*, vol. 7, no. 1, pp. 48–50, 1956.
[24] "S. N. A. Project. Stanford large network dataset collection. http://snap.stanford.edu/data/soc-LiveJournal1.html."
[25] "The Center for Discrete Mathematics and Theoretical Computer Science. http://www.dis.uniroma1.it /challenge9/download.shtml."
[26] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a social network or a news media?" in *Proceedings of the 19th international conference on World wide web*. ACM, 2010, pp. 591–600.
[27] "S. N. A. Project. Stanford large network dataset collection. http://snap.stanford.edu/data/com-Friendster.html."
[28] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "Accelerate large-scale iterative computation through asynchronous accumulative updates," in *Proceedings of the 3rd workshop on Scientific Cloud Computing Date*. ACM, 2012, pp. 13–22.
[29] L. Roditty and V. Vassilevska Williams, "Fast approximation algorithms for the diameter and radius of sparse graphs," in *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*. ACM, 2013, pp. 515–524.
[30] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson, "From think like a vertex to think like a graph," *Proceedings of the VLDB Endowment*, vol. 7, no. 3, pp. 193–204, 2013.
[31] U. Brandes, "A faster algorithm for betweenness centrality*," *Journal of mathematical sociology*, vol. 25, no. 2, pp. 163–177, 2001.
[32] M. Luby, "A simple parallel algorithm for the maximal independent set problem," *SIAM journal on computing*, vol. 15, no. 4, pp. 1036–1053, 1986.
[33] "https://en.wikipedia.org/wiki/Graph_coloring."
[34] D. J. Welsh and M. B. Powell, "An upper bound for the chromatic number of a graph and its application to timetabling problems," *The Computer Journal*, vol. 10, no. 1, pp. 85–86, 1967.
[35] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *ACM Sigplan Notices*, vol. 48, no. 8. ACM, 2013, pp. 135–146.
[36] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 456–471.
[37] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey, "GraphMat: High performance graph analytics made productive," *Proceedings of the VLDB Endowment*, vol. 8, no. 11, pp. 1214–1225, 2015.
[38] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and J. Shun, "Internally deterministic parallel algorithms can be fast," *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pp. 181–192, 2012.
[39] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 135–146.
[40] Y. Simmhan, A. Kumbhare, C. Wickramaarachchi, S. Nagarkar, S. Ravi, C. Raghavendra, and V. Prasanna, "Goffish: A sub-graph centric framework for large-scale graph analytics," in *European Conference on Parallel Processing*. Springer, 2014, pp. 451–462.
[41] C. H. Teixeira, A. J. Fonseca, M. Serafini, G. Siganos, M. J. Zaki, and A. Aboulnaga, "Arabesque: a system for distributed graph mining," in *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 2015, pp. 425–440.
[42] W.-S. Han, S. Lee, K. Park, J.-H. Lee, M.-S. Kim, J. Kim, and H. Yu, "TurboGraph: a fast parallel graph engine handling billion-scale graphs in a single PC," in *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2013, pp. 77–85.
[43] Y. Zhang, V. Kiriansky, C. Mendis, and M. Z. S. Amarasinghe, "Optimizing Cache Performance for Graph Analytics," *arXiv preprint arXiv:1608.01362*, 2016.
[44] K. Vora, G. Xu, and R. Gupta, "Load the Edges You Need: A Generic I/O Optimization for Disk-based Graph Processing," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, 2016.
[45] M.-S. Kim, K. An, H. Park, H. Seo, and J. Kim, "Gts: A fast and scalable graph processing method based on streaming topology to gpus," in *Proceedings of the 2016 International Conference on Management of Data*. ACM, 2016, pp. 447–461.
[46] L. Ma, Z. Yang, H. Chen, J. Xue, and Y. Dai, "Garaph: efficient gpu-accelerated graph processing on a single machine with balanced replication," in *2017 USENIX Annual Technical Conference (USENIX ATC17)*. USENIX Association, 2017.
[47] J. Zhong and B. He, "Medusa: Simplified graph processing on gpus," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 6, pp. 1543–1552, 2014.
[48] W. Zhong, J. Sun, H. Chen, J. Xiao, Z. Chen, C. Cheng, and X. Shi, "Optimizing graph processing on gpus," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 4, pp. 1149–1162, 2017.

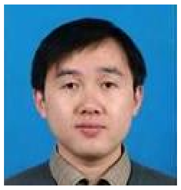
Zhiyuan Ai is a PhD student in Department of Computer Science and Technology, Tsinghua University, China. His research interests include graph processing and cloud computing. He received his B.E. degree from Harbin Institute of Technology, China, in 2013. He can be reached at: azy13@mails.tsinghua.edu.cn.

Mingxing Zhang received the PhD degree in computer science and technology from Tsinghua University, Beijing, China in 2017. His research interests include parallel and distributed systems. He received his B.E. degree from Beijing University of Posts and Telecommunications, China, in 2012. He can be reached at: zhang.mingxing@outlook.com.

Yongwei Wu received the PhD degree in applied mathematics from the Chinese Academy of Sciences in 2002. He is currently a professor in computer science and technology at Tsinghua University of China. His research interests include parallel and distributed processing, and cloud storage. Dr. Wu has published over 80 research publications and has received two Best Paper Awards. He is an IEEE senior member.



Xuehai qian received the PhD degree in Computer Science Department from University of Illinois at Urbana-Champaig, USA. Currently, he is an assistant professor at the Ming Hsieh Department of Electrical Engineering and the Department of Computer Science at the University of Southern California. His interests lie in the fields of computer architecture, architectural support for programming productivity and correctness of parallel programs.



Kang Chen received the PhD degree in computer science and technology from Tsinghua University, Beijing, China in 2004. Currently, he is an Associate Professor of computer science and technology at Tsinghua University. His research interests include parallel computing, distributed processing, and cloud computing.

Weimin Zheng received the BS and MS degrees, respectively, in 1970 and 1982 from Tsinghua University, China, where he is currently a professor of Computer Science and Technology. He is the managing director of the Chinese Computer Society. His research interests include computer architecture, operating system, storage networks, and distributed computing. He is a member of the IEEE.