Squeezing out All the Value of Loaded Data: An Out-of-core Graph Processing System with Reduced Disk I/O

Zhiyuan Ai, Mingxing Zhang, and Yongwei Wu, *Department of Computer Science and Technology, Tsinghua National Laboratory for Information Science and Technology* (*TNLIST*), *Tsinghua University and Research Institute of Tsinghua;* Xuehai Qian, *University of Southern California;* Kang Chen and Weimin Zheng, *Department of Computer Science and Technology, Tsinghua National Laboratory for Information Science and Technology* (*TNLIST*), *Tsinghua University, and Research Institute of Tsinghua*

https://www.usenix.org/conference/atc17/technical-sessions/presentation/ai

This paper is included in the Proceedings of the
2017 USENIX Annual Technical Conference (USENIX ATC '17).

July 12–14, 2017 • Santa Clara, CA, USA

ISBN 978-1-931971-38-6

Squeezing out All the Value of Loaded Data: An Out-of-core Graph Processing System with Reduced Disk I/O

[1]Zhiyuan Ai* [1]Mingxing Zhang* [1]Yongwei Wu [2]Xuehai Qian [1]Kang Chen [1]Weimin Zheng
[1]*Tsinghua University*† [2]*University of Southern California*

Abstract

The current primary concern of out-of-core graph processing systems is improving disk I/O locality, which leads to certain restrictions on their programming and execution models. Although improving the locality, these constraints also restrict the expressiveness. As a result, only sub-optimal algorithms are supported for many kinds of applications. When compared with the optimal algorithms, these supported 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 and execution model (e.g., process each loaded block at most once, neighborhood constraint) in a feasible manner, which enable efficient algorithms that require drastically less number of iterations. To demonstrate the ideas, we build 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 experiments show that the algorithms that can be only implemented in CLIP are much faster than the original disk-locality-optimized algorithms in many real-world cases (up to tens or even thousands of times speedup).

1 Introduction

As an alternative to distributed graph processing, disk-based single-machine graph processing systems (out-of-core systems) can largely eliminate all the challenges of using a distributed framework. These systems keep only a small portion of active graph data in memory and spill the remainder to disks, so that a single-machine can still process large graphs with the limited amount of memory. Due to the ease of use, several out-of-core systems have been developed recently [15, 26, 41]. 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 [41].

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 [15, 26, 41] 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 that would have otherwise occurred. The second is the *neighborhood constraint* that requires a single user-defined programming kernel to access only the 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 models, which assure the good disk I/O locality for the algorithms that they supported. 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.

As an illustration, the "at most once" requirement obviously 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

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

†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, Guangdong 518057, China.

bandwidth. As a consequence, current out-of-core systems only achieve *sub-optimal* performance.

In this paper, we propose CLIP, a novel disk-based graph processing system, in which supporting more efficient algorithms is the primary concern. We argue that out-of-core graph processing systems uniquely provide the opportunities to lift the restrictions of the programming and execution model (e.g., 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** accesses by allowing an "edge function" to update vertex properties that do not belong to the input 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. Without incurring development efforts, this method is vastly different from existing systems that load only needed part of vertices at a time (e.g., GraphChi, X-Stream, GridGraph).

Using this method, although the vertex data could reside in either memory or disk, Lin et al. [17] showed that the built-in caching mechanism of mmap is particularly desirable for processing real-world graphs, which often exhibit power-law degree distributions [12]. In such graphs, high-degree nodes tend to be accessed much more frequently than others and hence will always be cached in memory and result in good performance. Moreover, because the vertex data are typically much smaller than edge data but are accessed more frequently, our method is deemed to be a good heuristic in memory allocation that naturally reserves as much memory for vertices as possible. In fact, in our experiments, we *1)* test on many different real-world graphs that contain up to 6.6 billion edges; and *2)* modulate the maximum size of memory that the system is allowed to use for simulating the different size of available memory, from 32GB down to only 128MB (even 16MB for small graphs), by using cgroup. According to the results, CLIP is faster than any existing out-of-core systems on various memory limits.

The evaluation of our system consists of two parts. First, we evaluate the effectiveness of loaded data reentry, which can be applied to not only our system but also existing frameworks. According to our experiments, this simple technique can significantly reduce the number of required iterations for intrinsically iterative algorithms like SSSP and BFS, achieving up to 14.06× speedup. Second, we compare our novel beyond-neighborhood algorithms with prior ones on many important graph problems. We found that they can reduce the number of required iterations from 7∼6261 to only **one pass** for popular graph problems such as WCC (3.25×-4264× speedup) and MIS (20.9×-60× speedup).

2 Out-of-Core Graph Processing

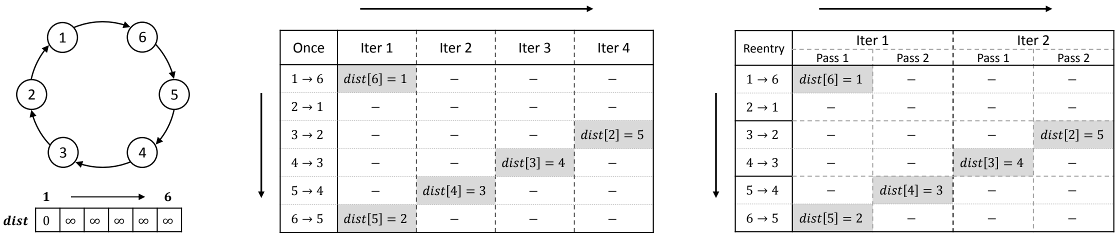
GraphChi [15] is the first large-scale out-of-core graph processing system that supports vertex programs. In GraphChi, the whole set of vertices are partitioned into "intervals", and the system only processes the related sub-graph of an interval at a time (i.e., only the edges related to vertices in 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 [15].

Some successor systems (e.g., X-Stream [26], GridGraph [41]) 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



(a) An example graph

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	$dist[5] = 2$	–	–	–

(b) The loaded block is processed only once

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	$dist[5] = 2$	–	–	–

(c) The loaded block is processed multiple times

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

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 natural, because a fully asynchronous graph processing would incur the tremendous amount 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 graph problems (e.g. SSSP, BFS) based on iterative improvement algorithms.

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 $dist[u]+1$. 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.

We argue that 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 block. 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. In the extreme case, if the user further enlarges the loaded data block to contain 6 edges, then only *one* iteration is needed. 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 [11, 29, 30] 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* [21], 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 [21, 23], 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 [8]. 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).

3.3 Limitations

Although the "beyond-neighborhood" algorithms offer significant performance improvements, it also becomes more difficult to infer the range of vertices that will be accessed by a user-defined function. As a result, it becomes more challenging to: *1)* selectively load vertex properties; and *2)* automatically parallelize the execution.

To address the first problem, our solution is to simply mmap all the vertices into memory. While counterintuitive, this straightforward method actually works quite well on many real-world scenarios. In our experiments, we test various data size (up to 6.6B edges) and memory limits (down to only 16MB for small graphs). Results show that our system largely outperforms existing ones in many real-world cases.

The reason of this phenomenon is two-fold. First, the size of vertices is usually considerably smaller than the size of edges but used much more frequently. Our method is deemed to be a good heuristic in memory allocation that naturally reserves as much memory for vertices as possible. Since the density of real-world dataset is usually larger than 30, in typical cases, our method could in fact keep *all* the vertices in memory. This behavior is even valid for industrial-grade workloads. Researchers in Facebook declared in their paper "One Trillion Edges: Graph Processing at Facebook Scale" [7] 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). This number means that most of the vertices can be cached in memory as the edges typically only need to be read in a stream fashion. This assumption is still valid after using reentry, because we only reentry the loaded edges.

Even more, as discussed in Lin et al. [17], the caching mechanism of mmap is particularly desirable for processing real-world graphs, which often exhibit power-law degree distributions. Our experiment results validate this assumption. Since these high-degree vertices are always cached in memory, accesses to their properties are cheap. In contrast, the other low-degree vertices may be swapped out if the memory limit is low, but they are accessed very infrequently.

As for the second problem, our observation is that: since the complexity of computation is quite low, disk I/O is the *real* bottleneck. It is also confirmed by our results in Section 5: the performance of our single-thread implementation can in fact match the multi-threaded **all-in-memory** systems and is significantly faster than prior multi-threaded *out-of-core* systems. The same phenomenon is also observed by many existing investigations [26], which conclude that there is no need of using multi-threading in an out-of-core environment. To be more general, we also provide a multi-threaded mode in CLIP, which requires users to use atomic operation if necessary. Based on our experience, the increased programming burden is quite limited (only requires the straightforward replacement of the original instruction by the atomic counterpart).

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 data graph and only the property of vertices can be modified. Figure 2 illuminates the main workflow of CLIP in detail. As for the computation, 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. This procedure is typically used to sort edges in grid order[1], where the length of grid is the page size. With this order, the accesses to the property of vertices show good locality. If this standard pre-processing is used, it is the **same** as GridGraph. But, by exposing this API to users, we provide more flexibility. 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).

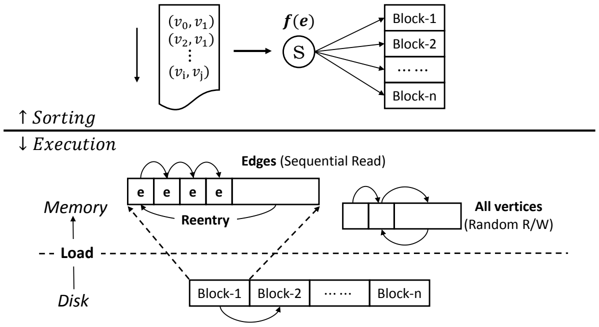


Figure 2: Main workflow of CLIP.

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 when most further local updating will be invalided by global updating.

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 [26, 41]. **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.

Table 1: Programming model of CLIP.

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

Our system also supports selective scheduling, which enables us to skip an edge or even a whole block if it is not needed. Specifically, through the *v_list* argument, $\mathcal{F}_e$ can both modify the property of an arbitrary vertex and set its activity. We define that *1)* an edge is inactive if its source vertex is inactive; 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.

4.3 Disk I/O

Although the bandwidth of disk is constantly improving, it still remains as the main bottleneck of out-of-core graph processing systems. Thus, in CLIP, we implement an overlapping mechanism by using a separate loading thread that continuously reads data into a circular buffer until it is full. Moreover, CLIP also enables selective scheduling to further improve the performance. This mechanism is implemented by maintaining the current activity of vertices with a bit-array. With this data structure, CLIP implements two kinds of skipping, namely edge skipping and block skipping. As we have mentioned in Section 4.2, for block skipping, an entire on-disk edge grid will be ignored when it does not contain any active edges (very easy to check bit-array since these source vertices are a continuous range). Moreover, in order to further enable edge skipping, one needs to use `Sort()` function to sort the input edges according to their source vertex. In that case, edges that have the same source vertex will be placed continuously and hence can be skipped at once if this source vertex is inactive (no need of checking the source ID for every edge).

4.4 Examples

To illustrate the usages of CLIP's API, this section presents the implementation of SSSP and WCC,

[1] Grid order means that the adjacent matrix of this graph is cut into grids and the edges belonging to the same grid are stored contiguously. Specifically, an edge (src, dst) is sorted by (src/grid_length, dst/grid_length, src, dst).

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 [5, 9]. 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.

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:
    $Exec(\mathcal{F}_e)$;
```

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. As we will show in Section 5.2.3, the value of MRT is important for achieving a good performance, but it is rather simple to choose an MRT that is good enough.

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.

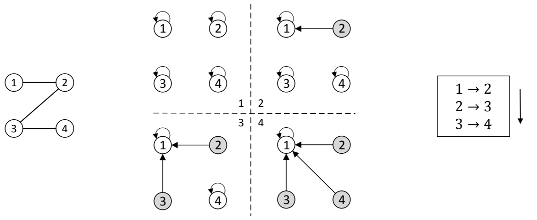
Algorithm 2 presents the code of our disjoint-set based WCC algorithm. Figure 3 gives an example. In our implementation, each vertex maintains a property *pa* that stores the ID of a vertex. If $pa[u] = v$, we name that the “parent” of vertex *u* is *v*. Vertex *u* is the representative of its subset 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.

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:
  $VMap(\mathcal{F}_v)$;
  $Exec(\mathcal{F}_e)$;
```

In Figure 3 (a), the graph has 4 vertices and 3 edges, the *pa* of every vertex is illustrated by arrows in Figure 3 (b). At the beginning of our algorithm, each vertex belongs to a unique disjoint subset. Hence, all arrows point to their starting vertex (1 in Figure 3(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 3(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 3(b)). Similarly, the arrow of vertex 4 is redirected to vertex 1 after reading edge $(3,4)$ (4 in Figure 3(b)). Eventually, all arrows point to vertex 1 and hence we found that there is only one weak connected component in the graph.

As one can imagine, this disjoint-set based algorithm always requires only one iteration to calculate WCC for an arbitrary graph, so that it leads to much less work than the original label-propagation based algorithm. But, a potential problem of this algorithm is that, when accessing the property of a vertex, it also needs to access its parent’s property (i.e., 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, it 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 their x index, which means that the grids are read in the order of "(0, 0), (0, 1), ..., (0, P-1), (1, 0), ..." if the graph edges are 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.



(a) An example graph (b) Execution process (c) Edge stream

Figure 3: WCC example.

5 Evaluation

In this section, we present our evaluation results on Clip and compare it with the state-of-art systems X-Stream and GridGraph (as they are reported to be faster than other existing out-of-core graph processing systems like GraphChi). We split all the benchmarks we tested into two categories by their properties and discuss the reason of our speedup respectively.

5.1 Setup

5.1.1 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 show that the single-thread algorithms implemented in Clip is even faster than the multi-threaded implementations in X-Stream and GridGraph, which can take advantage of at most 16 threads.

5.1.2 Benchmarks

We consider two categories of benchmarks. The first category is **asynchronous applications**, which includes SSSP, BFS and other algorithms like delta-based PageRank [37], diameter approximation [25], transitive closures [32], betweenness centrality [6], 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), which require users to develop new algorithms to achieve the best performance. One should notice that, for each application, we use either "reentry" **or** "beyond-neighborhood", so that there is no need for a piecewise breakdown of the performance gain.

5.1.3 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 2, from which we can see that the limit is down to only 16MB as the vertex number of LiveJournal is less than 5M.

Table 2: The real-world graph datasets. A random weight is assigned for unweighted graphs.

Graph	Vertices	Edges	Type	Threshold	
				external	semi
LiveJournal [3]	4.85M	69.0M	Directed	16MB	256MB
Dimacs [4]	23.9M	58.3M	Undir.	64MB	256MB
Twitter [14]	41.7M	1.47B	Directed	128MB	4GB
Friendster [2]	65.6M	1.8B	Directed	128MB	4GB
Yahoo [1]	1.4B	6.64B	Directed	4GB	8GB

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 testing Clip with **16** threads for asynchronous applications but only **one** thread for beyond-neighborhood algorithms.

5.2 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.

Table 3: 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 / -

5.2.1 Comparison

The results are presented in Table 3, 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.

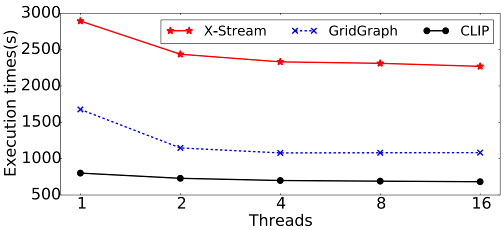


Figure 4: The scalability for SSSP on Twitter graph, evaluated in semi-external scenario.

5.2.2 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 4 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-thread baseline of GridGraph is inefficient. In fact, the single-thread CLIP is already faster than multi-thread version of GridGraph.

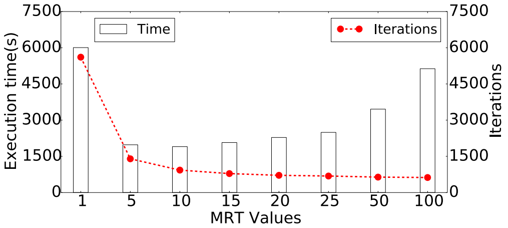


Figure 5: Execution time and required iterations for executing SSSP on Dimacs graph in the semi-external scenario, with different MRT values.

5.2.3 MRT

The value of Maximum Reentry Times (MRT) modulates the trade-off between global updating and local updating. As its effect depends not only on the property of input graph but also on the type of application, there isn't a rule for calculating the best MRT. But, according to our experiences, heuristically setting MRT to 5-10 is usually enough for producing a performance that is matchable with the best possible result (less than 4% difference). For example, all the values we reported in Table 3 is measured at "MRT = 5". The intuitive reason for this phenomena is that the diameter of a real-world graph is typically not large. Figure 5 shows the execution time and required iterations of SSSP on Dimacs graph with different MRTs. We see that both an excessively small MRT (e.g., =1) or an excessively large MRT ($\geq$20) are not helpful. When MRT is larger than 10, while the number of iterations is decreasing, the execution time will actually increase. The reason is that large MRT will lead to many useless iterations within each block, which increases the amount of calculation of CPU without propagating the updates to other blocks.

Table 4: Execution time (in seconds) on WCC and MIS. Format of this table is the same as Table 3. 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 2, 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 / -

5.3 Beyond-neighborhood

5.3.1 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 [19]. 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.

5.3.2 Comparison

Table 4 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 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, out 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. As we have explained in Section 3.3, 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 Discussion

6.1 Scope of Application

Although our "reentry" technique is quite simple, it essentially provides a **midpoint** between the fully synchronous algorithm and the fully asynchronous algorithm. It makes the convergence faster than fully synchronous execution but makes an implementation more "disk-friendly" than fully asynchronous execution (i.e. process once a block rather than once a vertex). As a result, all applications that can benefit from asynchronous execution can benefit from "reentry", because they are based on the same principle.

In contrast, the application of "beyond-neighborhood" does rely on the existence of such algorithms. But, according to our study, there are indeed a large set of applications can be optimized with our model. For example, finding WCC of a graph lies at the core of many data mining algorithms, and is a fundamental subroutine in graph clustering. Thus, our method can benefit not only WCC itself but also all these applications. Similarly, MIS shares a similar access pattern of many graph matching applications. In fact, the number of these so-called Graph Stream Algorithms is large enough for publishing a survey on them [8, 21].

Essentially, our "beyond-neighborhood" optimization fundamentally enhances the expressiveness of the vertex programs so that important graph operations like "pointer-jumping" could be implemented. A recent article [16] made the same observation but only discussed it in the context of Galois [24]. This paper shows that such more expressive programming model is not only applicable for in-memory but also feasible for out-of-core graph processing systems. Even more, we argue that the significant performance improvements that "beyond-neighborhood" can achieve also overshadows its limitation on applicability.

6.2 Programmability

McSherry et al. [22] have observed that the scalability of many distributed graph processing system is based on their inefficient single-thread implementation. As a result, they argue that specialized optimized implementations should be used in many real-world scenarios, which share the same principle as our system. However, different from their work that uses a set of distinct programs, Clip is a complete system that provides a general enough programming model.

The trade-off between more flexibility (potentially worse programmability) and better performance is well-known. Neighborhood-constraint systems choose one extreme of this spectrum, which provides the best programmability but worse performance. McSherry et al.'s work [22] and some others (e.g., Galois [24], smart algorithm in GoFFish [27], Polymer [35]) choose the other extreme. They provide only some basic functionalities (e.g., concurrent loop) or even barely anything. These methods can achieve the best performance, but impose a much larger burden on programmers.

In contrast, we believe that Clip is a sweet spot in the design space that is just right for out-of-core systems. The slight sacrifice of programmability is definitely worthwhile because this makes Clip up to tens and even thousands of times faster than existing systems. According to our evaluation, the programming model of Clip helps us to write all the programs described in this paper in less than 80 lines of codes, comparing to 1200 lines for the native algorithms (many lines of code are used for dealing with chores like I/O, partitioning, etc.).

6.3 Compared with In-memory System

Thanks to flexibility of Clip, its performance on many kinds of applications is matchable with in-memory systems. As an illustration, Table 5 presents the comparison between Clip (semi-external mode) and manually optimized algorithms that implemented in Galois. Since the loading of 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.

Besides Galois, GraphMat [28] is also an in-memory graph processing system that takes advantage from efficient matrix operations. According to our evaluation, GraphMat requires only 0.72s to calculate the WCC of LiveJournal, which is faster than both Galois and Clip (while it requires 9.78s for loading data). However, GraphMat employs a synchronous execution engine that enforces neighborhood constraint. Thus, for graphs that have a large diameter, its performance is poor. For example, GraphMat needs 6262 iterations (221.9s) to achieve the convergence of WCC algorithm on Dimacs (only 1 iteration and 1.35s are needed for Clip).

Table 5: Execution time (in seconds) for Clip and Galois. '-' designates out of memory.

	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

6.4 Concurrency

As mentioned in Section 5.2.2, users of Clip can enable multi-thread execution for applications that voluntarily obey the neighborhood constraint (e.g., SSSP). Specifically, for executing `VMap()` in parallel, the whole vertex set is split into equal intervals that are dispatched to different worker threads. Similarly, for executing `Exec()`, the loaded edge grid is further split and dispatched. With neighborhood constraint, the concurrency control can be implemented by fined-grained locking in a straightforward manner. However, although the locking mechanism can assure the correctness of our system, certain downsides of asynchronous execution still exist in CLIP, such as non-deterministic execution and unstable performance. However, asynchronous execution has been demonstrated to be able to accelerate the convergence of iterative computations [10].

Besides multi-threads, there are also some graph systems that support multi-tenant execution [7, 20]. Different from them, Clip is a single machine graph processing system and does not support multi-tenant execution, which is similar to prior systems [15, 26, 41]. Typically, multi-tenant is more useful for distributed systems that share the same cluster.

6.5 Evaluation on HDD

It is worth mentioning that, Clip can also achieve a good performance on slow storage devices (e.g., HDD). We evaluate Clip on a standard 3TB HDD and compare it with X-Stream and GridGraph. According to our evaluation, the average throughput of our HDD is about 150MB/s for sequential read. Table 6 shows the evaluation results under the semi-external scenario. Since the amount of loading data dominates the execution time, Clip can achieve a similar or even better speedup (5.59×-5999× for WCC, 2.32×-15.37× for BFS) with the evaluation on SSD.

Table 6: Execution time (in seconds) on HDD. '∞' means that the application does not finish after running 24 hours.

	LiveJournal	Dimacs	Friendster	Twitter	Yahoo
WCC					
X-Stream	128.1	15417	5219	2519	∞
GridGraph	34.68	16467	1314	785.9	8764
Clip	6.2	2.57	160.2	132.1	590.4
BFS					
X-Stream	53.25	16943	2566	1067	∞
GridGraph	34.28	12790	1431	604.8	22528
Clip	14.77	2659	93.1	217.3	8844

6.6 Preprocessing Time

Pre-processing is a necessary procedure for most (e.g., GraphChi, GridGraph, Clip) but not all (e.g., X-Stream) out-of-core graph processing systems. The preprocessing cost of Clip is similar to GridGraph, as they are almost the same. Moreover, although sometimes the pre-processing time is longer than the execution time, it is still worthwhile in terms of total execution time. For example, the total execution time (preprocessing+computation) of computing MIS on Friendster is 4867s for X-Stream and 3962.5s for GridGraph. In contrast, the total execution time of Clip is 145.3s for pre-processing and only 62.49s for computation, which in total is 207.79s. As we can see, the total execution time of Clip is 19.07× faster than GridGraph and 23.42× faster than X-Stream, not to mention that the preprocessing cost can be amortized by reusing the results.

7 Related Work

There are also many distributed graph processing systems. Pregel [20] is the earliest distributed graph processing system that proposes a vertex-centric programming model, which is later inherited by many other graph processing systems [12, 18, 26, 36, 40]. Some existing works [27, 31], such as Giraph++ [32], have suggested to replace "think as vertex" with "think as subgrapg/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 [13, 17, 38, 39]. 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 [17] 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 [34, 39] that aim to load only necessary data in an iteration, which can also reduce disk I/O. 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 method is enough for our case.

Some existing works [15, 33] are proposed to support evolving graphs, which is not currently supported in our system. But, although it is not discussed, the same mechanism for dealing with evolving graph in GraphChi can be added to Clip in a straightforward manner. To maintain the consistency of data, we reserve all the addition and deletion of edges within an iteration and only apply them in the interval between two iterations.

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

[2] S. N. A. Project. Stanford large network dataset collection. http://snap.stanford.edu/data/com-Friendster.html.

[3] S. N. A. Project. Stanford large network dataset collection. http://snap.stanford.edu/data/soc-LiveJournal1.html.

[4] The Center for Discrete Mathematics and Theoretical Computer Science. http://www.dis.uniroma1.it /challenge9/download.shtml.

[5] Richard Bellman. On a routing problem. *Quarterly of applied mathematics*, pages 87–90, 1958.

[6] Ulrik Brandes. A faster algorithm for betweenness centrality*. *Journal of mathematical sociology*, 25(2):163–177, 2001.

[7] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One trillion edges: graph processing at Facebook-scale. *Proceedings of the VLDB Endowment*, 8(12):1804–1815, 2015.

[8] Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. On graph problems in a semi-streaming model. In *International Colloquium on Automata, Languages, and Programming*, pages 531–543. Springer, 2004.

[9] Michael L Fredman. New bounds on the complexity of the shortest path problem. *SIAM Journal on Computing*, 5(1):83–89, 1976.

[10] Andreas Frommer and Daniel B Szyld. On Asynchronous Iterations. 1999.

[11] Harold N Gabow and Robert Endre Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of computer and system sciences*, 30(2):209–221, 1985.

[12] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos 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)*, pages 17–30, 2012.

[13] Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, Jeong-Hoon Lee, Min-Soo Kim, Jinha Kim, and Hwanjo 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*, pages 77–85. ACM, 2013.

[14] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a social network or a news media? In *Proceedings of the 19th international conference on World wide web*, pages 591–600. ACM, 2010.

[15] Aapo Kyrola, Guy Blelloch, and Carlos 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)*, pages 31–46, 2012.

[16] Andrew Lenharth, Donald Nguyen, and Keshav Pingali. Parallel Graph Analytics. *Communication of ACM*, 59(5):78–87, 2016.

[17] Zhiyuan Lin, Minsuk Kahng, Kaeser Md Sabrin, Duen Horng Polo Chau, Ho 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*, pages 159–164. IEEE, 2014.

[18] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[19] Michael Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM journal on computing*, 15(4):1036–1053, 1986.

[20] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[21] Andrew McGregor. Graph stream algorithms: a survey. *ACM SIGMOD Record*, 43(1):9–20, 2014.

[22] Frank McSherry, Michael Isard, and Derek G Murray. Scalability! But at what COST? In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, 2015.

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

[24] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 456–471. ACM, 2013.

[25] Liam Roditty and Virginia 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*, pages 515–524. ACM, 2013.

[26] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-Stream: edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 472–488. ACM, 2013.

[27] Yogesh Simmhan, Alok Kumbhare, Charith Wickramaarachchi, Soonil Nagarkar, Santosh Ravi, Cauligi Raghavendra, and Viktor Prasanna. Goffish: A sub-graph centric framework for large-scale graph analytics. In *European Conference on Parallel Processing*, pages 451–462. Springer, 2014.

[28] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R Dulloor, Michael J Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. GraphMat: High performance graph analytics made productive. *Proceedings of the VLDB Endowment*, 8(11):1214–1225, 2015.

[29] Robert E Tarjan and Jan Van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM (JACM)*, 31(2):245–281, 1984.

[30] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM (JACM)*, 22(2):215–225, 1975.

[31] Carlos HC Teixeira, Alexandre J Fonseca, Marco Serafini, Georgos Siganos, Mohammed J Zaki, and Ashraf Aboulnaga. Arabesque: a system for distributed graph mining. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 425–440. ACM, 2015.

[32] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. From think like a vertex to think like a graph. *Proceedings of the VLDB Endowment*, 7(3):193–204, 2013.

[33] Keval Vora, Rajiv Gupta, and Guoqing Xu. Synergistic Analysis of Evolving Graphs. *ACM Transactions on Architecture and Code Optimization (TACO)*, 13(4):32, 2016.

[34] Keval Vora, Guoqing Xu, and Rajiv 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.

[35] Kaiyuan Zhang, Rong Chen, and Haibo Chen. NUMA-aware graph-structured analytics. In *ACM SIGPLAN Notices*, volume 50, pages 183–193. ACM, 2015.

[36] Mingxing Zhang, Yongwei Wu, Kang Chen, Xuehai Qian, Xue Li, and Weimin Zheng. Exploring the hidden dimension in graph processing. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 285–300. USENIX Association, 2016.

[37] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. Accelerate large-scale iterative computation through asynchronous accumulative updates. In *Proceedings of the 3rd workshop on Scientific Cloud Computing Date*, pages 13–22. ACM, 2012.

[38] Yunming Zhang, Vladimir Kiriansky, Charith Mendis, and Matei Zaharia Saman Amarasinghe. Optimizing Cache Performance for Graph Analytics. *arXiv preprint arXiv:1608.01362*, 2016.

[39] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E Priebe, and Alexander S Szalay. FlashGraph: Processing billion-node graphs on an array of commodity SSDs. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 45–58, 2015.

[40] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A computation-centric distributed graph processing system. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)(Savannah, GA*, 2016.

[41] Xiaowei Zhu, Wentao Han, and Wenguang Chen. GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 375–386, 2015.