Wonderland: A Novel Abstraction-Based Out-Of-Core Graph Processing System

Mingxing Zhang[†§] Yongwei Wu[†] Youwei Zhuo[‡] Xuehai Qian[‡] Chengying Huan[†] Kang Chen[†]

[†]Tsinghua University* [‡]University of Southern California [§]Sangfor Technologies Inc.

Abstract

Many important graph applications are iterative algorithms that repeatedly process the input graph until convergence. For such algorithms, graph abstraction is an important technique: although much smaller than the original graph, it can bootstrap an initial result that can significantly accelerate the final convergence speed, leading to a better overall performance. However, existing graph abstraction techniques typically assume either fully in-memory or distributed environment, which leads to many obstacles preventing the application to an out-of-core graph processing system.

In this paper, we propose Wonderland, a novel out-of-core graph processing system based on abstraction. Wonderland has three unique features: *1)* A simple method applicable to out-of-core systems allowing users to extract effective abstractions from the original graph with acceptable cost and a specific memory limit; *2)* Abstraction-enabled information propagation, where an abstraction can be used as a bridge over the disjoint on-disk graph partitions; *3)* Abstraction-guided priority scheduling, where an abstraction can infer the better priority-based order in processing on-disk graph partitions. Wonderland is a significant advance over the state-of-the-art because it not only makes graph abstraction feasible to out-of-core systems, but also broadens the applications of the concept in important ways. Evaluation results of Wonderland reveal that Wonderland achieves a drastic speedup over the other state-of-the-art systems, — up to two orders of magnitude for certain cases.

*M. Zhang, Y. Wu, C. Huan, and K. Chen are with the Department of Computer Science and Technology, Graduate School at Shenzhen, Tsinghua National Laboratory for Information Science and Technology (TNLIST), Tsinghua University, Beijing 100084, China; M. Zhang is also with Sangfor Technologies Inc.

ASPLOS '18, March 24–28, 2018, Williamsburg, VA, USA

ACM ISBN 978-1-4503-4911-6/18/03…$15.00
https://doi.org/10.1145/3173162.3173208

CCS Concepts • **Computer systems organization** → **Multicore architectures**; • **Hardware** → *External storage*;

Keywords graph computing, abstract, out-of-core

1 Introduction

As an alternative of distributed graph processing, disk-based single-machine graph processing systems (a.k.a., out-of-core systems) keep only a small portion of active graph data in memory and spill the remainder of a large graph to disks, which make practical large-scale graph processing available to anyone with a modern PC [21]. More importantly, it is also demonstrated that the performance of an out-of-core system on a single machine can even be competitive compared with a distributed graph processing framework using hundreds of cores [45]. Due to the ease of use and promising performance, several recent out-of-core systems have been developed [18, 21, 23, 30, 42, 45].

Although the slow random disk I/O is usually the main concern of designing out-of-core systems, recent works [45] demonstrated a great utilization of disk's sequential bandwidth, making the slow random disk I/O no longer a performance bottleneck. In contrast, the convergence speed, which is ultimately determined by the speed of information propagation in the graph, still remains as the major factor affecting performance. In our experiments, GridGraph [45] usually needs **more than 10 minutes** to find the shortest path between two vertices of a 17GB power-law Twitter graph [20] using different memory size limits, e.g., 512MB, 4GB and 16GB. We will show that it is possible to reduce the execution time of the *same computation* **to 72.9s/45.6s/9.31s, — 9.3x/14.1x/66.5x faster** with 512MB/4GB/16GB memory limits, using the *same machine*. Clearly, the drastic speedup has to come from the fundamental improvement on *execution efficiency*, which is the focus of this paper.

The key technique enabling this improvement is *graph abstraction*, which is a concise lossy representation of the original graph. This technique is originally applied in visualization systems, as it can provide a high-level understanding of the information encoded in large graphs [35]. Interestingly, it has also been shown that graph abstraction can also benefit graph processing by bootstrapping an initial result that can accelerate the final convergence speed [6, 7, 14–17, 19, 22, 25, 27, 28, 31, 43]. However, applying graph abstraction to out-of-core systems is still an open problem. The fundamental obstacle is the *performance challenge*. To the

best of our knowledge, most existing techniques assume that the original graph can be fully contained in memory, hence directly applying them to out-of-core systems unavoidably incurs excessive random disk I/O. The problem is exaggerated considering *1)* the complex graph transformation operations they use; and *2)* the potentially multiple passes of reading the original graph.

Another obstacle is the *programmability challenge*, which is not limited to out-of-core systems. Specifically, due to the possibility of adding vertices/edges not existing in the original graph to the abstraction, programmers are required to design application-specific algorithms to transform the result on abstraction to an initial result on the original graph, which are demonstrated to be error-prone [19].

This paper attempts to apply graph abstraction to out-of-core systems. We propose a general abstraction generating procedure which *1)* produces *application-independent* abstractions; *2)* only reads *one pass* of the original graph data; and *3)* enables users to set an upper bound of memory usage. Therefore, we fundamentally advance the state-of-the-art by solving both performance and programmability challenge.

Even more, with an out-of-core system equipped with graph abstraction, we seek new opportunities to fully unleash the potential of graph abstraction. The insight is that, as a concise lossy representation, graph abstraction can *encode information* that captures certain property of the original graph. We propose two novel abstraction-based techniques that are naturally applicable to out-of-core systems.

The first technique is *abstraction-enabled information propagation*. Existing out-of-core graph processing systems typically partition the data into disjoint on-disk graph partitions [21, 45]. The disjointness increases the number of iterations because each edge is processed at most once per iteration. As a form of information encoder, we propose to use graph abstraction as a "bridge" among the disjoint graph partitions so that the information can be propagated across them more thoroughly in the same iteration, leading to faster convergence speed.

The second technique is *abstraction-guided priority scheduling*. A common pattern of existing out-of-core graph processing system is that they iterate the graph in a specific pre-defined order. Although such order may not be optimal, it is difficult to decide an alternative order that is likely better. With the information encoded in the graph abstraction, the system could preferentially loading the graph partition that may produce better immediate results, again leading to less number of iterations. The integration of this technique to the existing systems is straightforward, but it is graph abstraction that opens such a new opportunity.

In essence, the abstraction-based optimization techniques *generalize the current out-of-core systems*, which could be considered as having "empty" abstraction. Therefore, they cannot benefit from the additional information encoded in graph abstraction that enables faster convergence. For this reason, we believe that abstraction is not only practical, but also should be a *standard component* for any out-of-core systems.

Based on the novel techniques, we build *Wonderland*, a novel abstraction-based out-of-core graph processing system. We divide the applications supported in Wonderland into two categories: *1) any-path* queries that can be answered by any path between two vertices; and *2) all-path* queries that can only be answered by checking all the paths between two vertices, given certain effective pruning rules. We evaluate the two kinds of queries separately and the results show that Wonderland achieves significant speedups over the state-of-the-art systems, — up to two orders of magnitude for certain cases. The aforementioned impressive shortest path (an example of all-path query) results on Twitter graph is achieved by Wonderland, in which graph abstraction fundamentally improves execution efficiency. Moreover, we provide the detailed piece-wise breakdown of speedups from *each* proposed technique, indicating that the overall acceleration is due to the combination of all three optimizations.

2 Background

2.1 Out-of-core Graph Processing

GraphChi [21] 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" processed one by one in an iteration. For each interval, GraphChi ensures that all edges in an interval are stored in only a few contiguous regions on the disk. As a result, GraphChi requires a small number of non-sequential disk accesses during processing and achieves good performance that is competitive to a distributed graph processing system [21].

Different from GraphChi, X-Stream [30] provides an edge-centric programming model, in which the smallest computation granularity is an edge and its two vertices. X-Stream uses a *shuffle* procedure that executes an external sort to connect its two computation phases: *1)* a *scatter* phase that reads all edges and outputs all updates in a stream fashion; and *2)* a *gather* phases that applies the updates.

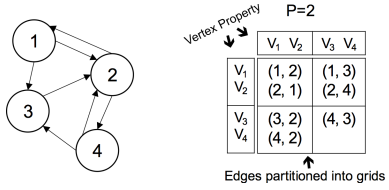


Figure 1. A graph partitioned into 2×2 grids.

Based on X-Stream, GridGraph [45] proposes a 2D-partitioning method to improve data locality by splitting edge blocks. As illustrated in Figure 1, GridGraph also partitions the vertices into P intervals, where each interval contains vertices within a contiguous range. However, in GridGraph, the edges are partitioned in $P \times P$ grids rather than only P shards as in GraphChi. Each *grid[i, j]* contains edges that start from vertices in interval i and end in vertices of

interval j. With this data model, the computation model of GridGraph is similar to X-Stream. Users can define an edge function that reads the property of an edge and its source vertex and updates its destination vertex. The only difference is that, in GridGraph, an edge's update can be applied directly, instead of outputted in a stream fashion. As a result, in processing *grid[i,j]*, *1)* the property of the edges in this grid and the property of vertices in interval i are read; and *2)* the property of vertices in interval j are written. The good disk locality is achieved because all accessed data are stored contiguously so that they could be sequentially loaded into memory. As a result, GridGraph is shown to be much faster than X-Stream and GraphChi.

However, despite the performance differences, existing out-of-core systems share two common patterns. *1) Disjoint Partitions:* Existing systems partition the graph data into *disjoint* parts so that each part can be loaded sequentially into memory. Vora et al. [38] try to enhance GraphChi by dynamically adjusting the intervals so that only needed edges are loaded into memory. However, the disjoint property still applies. We find that this property ensures disk locality at the expense of limiting convergence speed. The fact that each edge is processed at most once per iteration prevents more thorough information propagation in different parts. *2) Fixed Processing Order:* Existing systems assume a *fixed order* in loading and processing the disjoint data partitions. Although such order may not be optimal, it is difficult to come up with an alternative order that could be better due to the lack of guidance information.

2.2 Graph Abstraction

Graph abstraction is a concise lossy representation of a graph, which can typically be stored in memory and accelerate graph processing. Typically, the graph computation is first applied on the small graph abstraction. Then, the result can be transferred into an *initial result* on the original graph which leads to faster convergence to the final precise result.

Specifically, six different transformation rules are defined in [19] to reduce the size of original graph by removing or merging vertices and edges. Figure 2 presents three examples: *1)* removing a vertex if it has no incoming/outgoing edges; *2)* transforming a vertex y that has a single incoming edge (x, y) and a single outgoing edge (y, z) to an edge (x, z); *3)* for a vertex v with high number of incoming edges, merging the source vertices of those incoming edges with v.

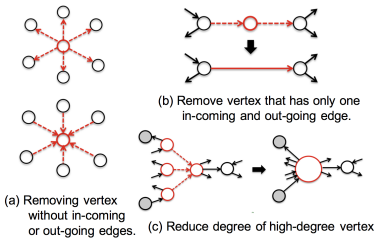


Figure 2. Operations in [19].

Besides the transformation rules, the properties that can be preserved after applying these transformations are also analyzed. These properties can be used to guide the design of application-specific *result transformation procedure* that transfers the result on the graph abstraction to an initial result on the original graph. According to [19], a good initial result can lead to 1.25×-2.14× speedup for various algorithms (e.g., SSSP, PageRank, Connectivity).

2.3 Open Problem: Applying Graph Abstraction to Out-Of-Core Systems.

Despite the promise of graph abstraction, applying it to out-of-core systems is still an open problem. The first challenge is *performance*. To the best of our knowledge, most existing techniques assume that the original graph can be fully contained in memory. Hence, directly applying them to out-of-core systems unavoidably incurs excessive random disk I/O. The problem is exaggerated considering *1)* the complex graph transformation operations they use, such as merging vertices into super vertex; and *2)* the potentially multiple passes of reading the original graph.

Another challenge is *programmability*, which is not limited to out-of-core systems. Specifically, due to the possibility of adding vertices/edges not existing in the original graph to the abstraction (e.g., rule 2 and 3 in Figure 2), programmers are required to design a transformation procedure that transfers the result on the graph abstraction to an initial result on the original graph. The algorithm in such transformation is typically *application-specific*, which may even require a redesign of the original graph algorithm. According to [19], even different algorithm implementations of the same graph application may enforce different requirements on the abstraction generation method, and a mismatch will lead to incorrect answers. In the SSSP example in [19], a subtle initialization method (affecting only one line of code) has a decisive impact on the correctness of the final results.

The primary goal of this paper is to enable graph abstraction in out-of-core systems without error-prone transformation. Our key observation is that it is possible to use abstractions that *consist of only vertices and edges existing in the original graph*. This can be implemented by simply selecting a subset of the original graph's edge set as its abstraction, making it possible to only perform a *single pass* read of original graph during graph abstraction generation. This method produces *application-independent* abstractions, avoiding error-prone transformations; and allows users to set an upper bound of memory usage, which naturally fits the nature of out-of-core systems.

Moreover, we seek *additional opportunities* to fully unleash the potential of graph abstraction. The key insight is that, as a concise lossy representation, graph abstraction can *encode information* that captures certain property of the original graph. It can facilitate information propagation and more efficient execution scheduling.

3 Wonderland

This section describes the design of *Wonderland*, a novel out-of-core graph processing system based on abstraction.

3.1 System Overview

The data model of Wonderland is the same as GridGraph, i.e., a directed data graph that has mutable vertex property and read-only edges. For undirected graphs, the edge between two vertices is represented by two reverse edges. As reported by previous works [29, 30, 45], the immutability of edges does not affect the expressiveness of a framework.

The workflow of Wonderland is shown in Figure 3, which is largely different from existing works. The central piece in this workflow is an abstraction of the original large graph. It has two phases: *1)* abstraction generation, which only needs to be done once before the second phase, and *2)* query processing. In Wonderland, the abstraction is simply a subset of original graph's edge set that contains at most X edges.

In query processing, the generated graph abstraction is reused for queries of the same or different applications. Specifically, the system maintains an in-memory sub-graph of the original graph (i.e., `in_memory_graph`), which initially contains only the abstraction (line 10). In each iteration[1], the user-defined processing function is used to update vertices contained in this sub-graph (line 12). This procedure is repeated until a certain convergence condition is reached. In the first iteration, the execution of line 12 is to process the query only on the abstraction, which can be considered as the step to bootstrap an initial result. The abstraction-enabled information propagation and the abstraction-guided priority scheduling can be expressed naturally in this workflow.

Abstraction Generation The first phase of Wonderland is to generate a graph abstraction, initiated as an empty set (line 2), and is iteratively updated. For each abstraction generation iteration (line 3-6), the system expands the current abstraction with several edges from the original graph (line 4), and then selects at most X edges from the expanded abstraction to remain in `abstract` (line 5). The edges not selected (stored in `deleted`) are dumped to disk in a stream fashion (line 6). This generation procedure is especially suitable for out-of-core systems because: *1)* in each iteration, only *a part* of graph (i.e., the expanded abstract) is loaded into memory, based on which, a *limited view* of the graph, edges are selected; *2)* the original graph is read only *once*; and *3)* the generated abstraction only consists of *edges in the original graph.*

Abstraction-Guided Information Propagation The out-of-core graph processing systems partition the graph into disjoint parts.For clarity, we use the term *round* to refer to the processing of *all* parts (i.e., the whole graph), while the term *iteration* refers to processing of *one* part. In current systems (e.g. GridGraph), the information is propagated in a limited fashion during a round since each edge is only processed once. This eventually leads to slower convergence. In Wonderland, graph abstraction serves as a *bridge among these disjoint parts* by processing edges in the abstraction multiple times (they are combined with different parts to form `in_memory_graph` in each iteration). The more thorough information propagation speedups the convergence. To achieve this goal, at the end of every iteration, a part of the on-disk graph is chosen (defined by `Choose` function) to be loaded into memory (line 13). Then, a *new* in-memory sub-graph is generated using *both* the abstraction and the chosen edges (line 14). Compared to processing only the disjoint parts, Wonderland allows a common abstraction to be shared among different iterations and hence enables information propagation between different parts in a round through the abstraction.

Abstraction-Guided Priority Scheduling With Wonderland's workflow, the order of processing the edge blocks can be specified in `Choose` function. This could be used to implement the abstraction-guided priority scheduling. The users specify the scheduling policy in this function to determine the proper scheduling order based on the information encoded in the abstraction. In Section 4.2, we will explain a simple example showing how to define the priority using on a specific graph application (i.e., shortest path).

```
1  // Abstraction Generation
2  abstraction = on-disk = {}
3  while not graph.empty()
4    abstraction = {abstract, graph.PopN(...)}
5    abstract, deleted = Select(abstract, X)
6    on-disk = {on-disk, deleted}
7
8  // Query Processing
9  foreach query
10  in_memory_graph = {abstract}
11  while not converge
12     Process(query, in_memory_graph)
13     load = Choose(on-disk)
14     in_memory_graph = {abstract, load}
```

Figure 3. An Overview of Wonderland's Workflow.

Although the workflow of Wonderland is largely different from existing frameworks, we intend to ensure the same level of programmability as existing graph processing systems, while, at the same time, providing excellent speedup. To this end, we need to consider several problems: *1)* how to implement the `Select` function that provides flexible options to generate abstraction for different applications; *2)* how to reduce random disk accesses; *3)* how to implement the `Choose` function to decide which part of the remained graph should be loaded into memory next; and *4)* how to implement the `Process` function in a user-friendly manner (e.g., providing the familiar vertex program interface). In the following, we discuss how these problems are solved in Wonderland, which leads to the refined workflow demonstrated in Figure 4. A detailed case study based on Wonderland is presented in Section 4.

3.2 Abstraction Generation

In Wonderland, the abstraction is generated incrementally through multiple abstraction generation iterations, each of

[1]Here, the meaning of "iteration" is different from GridGraph: each iteration in Figure 3 (line 11-14) processes a sub-graph.

which reads a number of edges that may be included in the abstraction. With limited amount of memory, users specify two parameters: X indicates the maximum size of the abstraction, and B indicates the number of edges read in each iteration. They should be chosen such that $X + B$ edges could be held in the machine's memory.

In principle, a smaller B leads to better abstraction but longer preprocessing time. Based on our experiments, a proper choice is to have roughly $B/X = 1/4$. X is a key factor determining performance. A small abstraction cannot significantly reduce the number of iterations while a large abstraction leads to a less number of iterations with longer execution time of each. Therefore, X should be carefully chosen. We will discuss this trade-off in Section 4.7.

The low-level API provided to users for abstraction generation is: `Select(vector<Edge>& abstract, size_t X)`. This `Select` function can use an arbitrary method to sort the edges contained in the input `abstract` vector. After sorting, the first X edges remained while the other edges will be dumped to disk (line 12-15 of Figure 4).

```
1  // Input
2  X = size of abstraction
3  B = the number of edges loaded per iteration
4  S = maximum size of each edge grid
5  W = width of grid
6
7  // Generating Abstract
8  abstract = vector<Edge>()
9  on-disk = fstream(...)
10 while not graph.empty()
11   abstract = {abstract, graph.PopN(B)}
12   abstract, deleted = Select(abstract, X)
13   on-disk.write(deleted)
14
15 // Remapping and Partitioning
16 abstract, grids = Partition(abstract, on-disk)
17
18 // Processing Querys
19 foreach query
20  in_memory_graph = {abstract}
21  worklist =
22    BootstrapWorklist(in_memory_graph, query)
23  while not converge
24     // Worklist-based processing
25     while not worklist.empty()
26       u = worklist.pop()
27       for e in in_memory_graph.loaded_edges(u)
28         ProcessEdge(u, e)
29         Append worklist accordingly
30       Update priority of grids accordingly
31     // Bootstrap next iteration
32     {grid1, grid2, ...} = Choose(grids, B)
33     in_memory_graph =
34       {abstract, grid1, grid2, ...}
35     worklist =
36       BootstrapWorklist(in_memory_graph, query)
```

Figure 4. Refined Workflow of Wonderland.

To ensure good programmability, Wonderland provides two higher-level APIs. The first is an edge-priority-based selection function. Specifically, users can define an `EdgePriority(Edge e)` function, whose input is an edge and output is this edge's priority. With this, the `Select` function chooses X edges with highest priorities.

Moreover, Wonderland provides a built-in function that generates an abstraction that contains as few weakly connected components as possible. The insight is that, since the abstraction can be used as bridges that immediately propagate updates from one disjoint part of the graph to others, an abstraction should connect as many vertices as possible. To implement this build-in function, a disjoint-set [10] data structure is used to automatically track the connectivity between vertices. Specifically, the edges will be iterated twice (both in the user-defined priority order). In the first iteration only edges that can merge two unconnected weakly connected component will be added to the remained edges, just like the procedure of the Kruskal's algorithm [8]. Then, in the second iteration, the edges that are not selected in the first iteration are added according to their priority order if the size of abstraction is still less than X.

3.3 Reducing Random Disk Accesses

In Wonderland, the property of all the vertices and edges are contiguously stored on two disk files, which are mapped into memory via `mmap` provided by OS. Similar to GridGraph, edges are partitioned into *grids*. With `mmap`, there are no explicit disk reads. However, since the vertex and edge file may be larger than memory, the performance can be severely affected by many random disk accesses if vertex and edge data are not properly organized. We consider the random disk accesses in two scenarios.

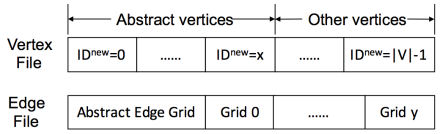


Figure 5. Disk Data Organization of Wonderland. Each edge grid is a sub-graph stored in CSR format, the grids can be ordered in arbitrary order.

Abstraction Different from existing out-of-core systems, random disk I/O could be incurred when accessing edges and related vertices in the abstraction, which could be in any part of a graph. A straightforward way to reduce the random edge accesses is to store them contiguously on disk. As shown in Figure 5, the edges of both abstraction and grids are stored contiguously in a single file `mmap`-ed into memory. The edges of abstraction are stored in the head of this file because the abstraction is accessed together with all other grids (see line 35 in Figure 4). Since the size of abstraction (e.g., X edges) is smaller than the machine's memory, typically the edges in abstraction are cached in memory.

Although the number of vertices contained in an abstraction is smaller than X, they may incur random accesses due to the non-contiguous IDs. To solve this problem, we `remap` the original vertex ID to ensure that the accesses to vertices in the abstraction are re-directed to the head of vertex property file (as illustrated in Figure 5). For example, consider

four vertices and only vertex 2 and vertex 4 are in the abstraction, the originally vertices ID=[1,2,3,4] are remapped to ID^{new}=[2,4,1,3] to ensure that: *1)* all the vertices contained in the abstraction have smaller IDs than the other vertices; and *2)* the original order is preserved for vertices contained (or not contained) in the abstraction. In Wonderland, this remapping procedure is implemented by a binary search on an in-memory vector which contains all vertices in abstraction. For each vertex X, this binary search returns that *1)* whether it is remained in the abstract or not; and *2)* there are Y vertices remained in the abstract that have smaller ID than X. Then, if this vertex is remained, this vertex's ID is remapped to Y. In contrast, if it is not remained, its ID is remapped to $X - Y$ plus the number of vertices remained in the abstract. This procedure incurs only one additional pass of all the edges with $O(X)$ space complexity.

Other Edges The edges not in abstraction are included in other grids. During processing, the edges in a grid are loaded into memory, extending the abstraction to form a new in-memory sub-graph. All edges and vertices of this sub-graph may be accessed in this iteration. Similar to GridGraph [45], the disk access locality is ensured by the grid organization. Specifically, with a user-defined grid width W, the whole graph is partitioned into $\lceil |V|/W \rceil \times \lceil |V|/W \rceil$ grids.

In Wonderland, the abstraction can be considered as a special grid, but not further partitioned. To make sure the in-memory sub-graph can fit in memory, users define a parameter S based on memory size to cap the maximum number of edges in a grid. If a grid contains X edges, it is further partitioned into $\lceil X/S \rceil$ grids. This further partitioning can be performed randomly or based on the priority of the edges assigned by the aforementioned `EdgePriority` function. The edges in each grid are contiguously stored in the edge file and is organized in Compressed Sparse Row (CSR) format, allowing users to locate all outgoing edges of a vertex in $O(1)$ complexity. Moreover, we require that a grid must be loaded as a whole: an edge is loaded into memory if and only if all the edges belonging to the same grid are also loaded. It ensures that the loaded edges (vertices) are store contiguously in the edge (vertex) file.

With the principle of radix sort, the grid partitioning procedure requires only **two** passes of the edges. The first pass counts the edges of each grid and the second pass writes edges to its corresponding locations. Moreover, the first pass of this partitioning procedure can be merged with the remapping procedure. Therefore, the whole preprocessing of Wonderland requires only three passes of the edges: one for selecting abstraction and two for remapping and partitioning.

3.4 Execution

Wonderland provides both low- and high-level APIs for users to express algorithms. At *low-level*, users could define `Process(Graph& g)` function, where `g` contains a member `g.loaded_grids`, a vector of all the loaded grids (abstraction is considered as a special grid). For each grid `g.grids[i]`, the smallest and largest ID of source vertices are stored in its member function `StartVertex()` and `EndVertex()`. For a vertex u, one can access its outgoing edges by iterating between `StartEdge(u)` and `EndEdge(u)`.

```
1  func VertexProgram(Graph& g, Index u)
2    // Iterating loaded edges
3    foreach grid in g.loaded_grids
4      foreach edge in [grid.StartEdge(u),
5                       grid.EndEdge(u))
6        ProcessEdge(g.Vertex(u), edge,
7                    g.Vertex(edge.destination))
8    // Updating priority of grids
9    foreach grid in g.all_grids
10     if u >= grid.EndVertex(): continue
11     if u < grid.StartVertex(): continue
12     UpdatePriority(g.Vertex(u), g.Priority(grid))
```

Figure 6. Typical Procedure of Processing a Vertex.

To ensure good programmability, Wonderland provides a *high-level* "Think Like a Vertex"-based interface similar to existing graph processing systems. Figure 6 shows the typical procedure of processing a vertex, which is a refined version of line 29-32 in Figure 4. In this program, `g.Vertex(u)` is used to get the writable reference of u's vertex property, and `g.Priority(grid)` is used to get the reference of the grid's priority. As one can see, the two-level *foreach* loop executes almost the same edge function as GridGraph, the only difference is that more than one grids can be loaded at the same time and the abstraction grid is always loaded. The second one-level *foreach* loop is optional. One can use it to update the the grids' priority. An example of this vertex program is given in Section 4.2.

In order to automatically parallelize the execution of user-defined vertex program, we implement a multi-threaded executor based on the low-level APIs provided by Galois [29]. Essentially, Galois provides a parallel *for_each* function, which uses multiple threads to concurrently consume the items contained in the work list. It also enables users to dynamically insert items to this work list during processing. This workflow is the same as our processing procedure shown in line 27-32 of Figure 4. Thus, it is easy to build algorithms on top of *for_each* function in Galois. In essence, we reuse the execution engine of Galois but replace its graph data organization part.

3.5 Optimization

Wonderland's low-level APIs enforce only a few restrictions to the program: for each process iteration, *1)* several edge grids are loaded to construct an in-memory sub-graph; and *2)* only the vertices and edges in this sub-graph can be accessed during this iteration. As a result, users can add various optimizations on top of Wonderland.

Among the possible optimizations, the activity optimization is used very frequently in graph systems. Thus, we add some syntax sugars to facilitate its usages. Specifically, we

add a bitmap file that is also `mmap`-ed into memory, and provide user APIs to set or clean a vertex's activity via the `Graph` variable. This information allows to skip certain grids if they contain no activated source vertices, which can be easily integrated into the abstraction-based priority scheduling.

4 Case Study: Shortest Path

To demonstrate the usages of Wonderland, this section presents a case study of processing shortest path queries.

4.1 Problem and Basic Algorithm

The input of a shortest path query is a weighted graph and two vertices *src* and *dst*. The output is the distance of the shortest path from *src* to *dst*. Edge weights can be negative, but we assume that there are no negative weight cycles. Similar to most algorithms that calculate shortest path, we attach a *dist* property to every vertex to hold the distance of the shortest known path from *src* to that vertex. This property is initialized to 0 for *src* and ∞ for all the other vertices, which will be iteratively updated with the relaxation operation: given an edge *(u,v)*, if $dist[u] + w[u,v] < dist[v]$, then $dist[v]$ is updated to $dist[u] + w[u,v]$. When no further relaxations can be performed, the resulting *dist* properties are the shortest distances from *src* to all vertices in V.

4.2 Implementation

The vertex program presented in Figure 7 is the basic implementation of shortest path query in Wonderland. It consists of two parts: *1)* iterating all the loaded outgoing edges of *u*, performing relaxation if necessary; and *2)* updating the priority of every on-disk edge grid.

Updating Vertices In the first part (line 4-15), the implementation of Wonderland's relaxation operation is almost the same as existing graph systems. For a vertex, the program simply iterates all outgoing edges (line 6) in all loaded grids (line 5), performs the relaxation if `new_dist` is smaller than `dst_dist`. The `Activate()` method of `Graph` variable `g` is used to set the activity bit of the edge's destination vertex if a relaxation is performed. One can optionally append the relaxed vertex into work list, or otherwise, all the loaded vertices are processed once.

Grid Priority For shortest path, we *estimate* the lower-bound of distance from *src* to *dst* through a path that uses at least one edge in the grid. Clearly, a grid with smaller lower-bound should have higher priority, thus we use the negative of this value as its priority.

We define `min_expect[i,dst]` for a grid[i] and each *dst*, which is a pre-calculated value that is the lower bound of a path starting from an edge in grid[i] to *dst*. Consider a vertex *u*, which is the source of an edge in grid[i], suppose the current shortest path from *src* to *u* is `Vertex(u).dist`. If `(Vertex(u).dist+min_expect[i,dst])` is smaller than the current priority, the priority of the grid should be updated to `-(Vertex(u).dist+min_expect[i,dst])`. Essentially, we use the sum as an estimation of the shortest path from *src* to *dst*, it is *approximate* because *u* may not be the source of edge corresponding to `min_expect[i,dst]`. However, it is always correct as the actual distance through *u* can only be larger.

```
1  func VertexProgram(Graph& g, Index u)
2    float src_dist = g.Vertex(u).dist
3
4    // Iterating loaded edges
5    foreach grid in g.loaded_grids
6      foreach edge in [grid.StartEdge(u),
7                       grid.EndEdge(u))
8        float new_dist = src_dist + edge.weight
9        float& dst_dist =
10             g.Vertex(edge.destination).dist
11       if new_dist < dst_dist
12         dst_dist = new_dist
13         g.Activate(edge.destination)
14         // Optional
15         Worklist.push(edge.destination)
16
17   // Updating priority of grids
18   foreach i in [0, g.all_grids.size())
19     Grid& grid = g.all_grids[i]
20     if u >= grid.EndVertex(): continue
21     if u < grid.StartVertex(): continue
22     float new_priority =
23              -(src_dist + min_expect[i,dst])
24     if grid.priority < new_priority
25       grid.priority = new_priority
```

Figure 7. Pseudocode of shortest path's vertex program.

Based on the definition, we need to compute a `min_expect[i,dst]` for each *dst*, which is not practical. However, due to the nature of approximation, we can significantly reduce the cost of pre-calculating the lower bound: each grid[i] may keep only one `min_expect[i]`, which is the minimum edge weight among all edges in grid[i]. Indeed, this lower bound is quite "loose", because it does not even consider the path finally reaching *dst*. However, we observe that even this simple method could lead to much better grid scheduling decision. The grid priority updating procedure is indicated in line 17-25 of Figure 7. For every scheduled vertex *u*, we iterate all the grids (rather than only loaded grids) and update the grid's priority if: *1)* the grid may contain an edge whose source vertex is *u* (line 20-21); and *2)* the shortest path from *src* to *u* (`src_dist`) may lower the estimated lower-bound distance defined earlier: `-(src_dist+min_expect[i])` is larger than the current `grid.priority`, line 22-25. Note that iterating all grids does not lead to extra overhead, because we only access the vertex boundary and priority.

It is important to understand the approximative nature of the grid priority: besides the conservative estimation of the lower-bound distance as we have discussed, assuring `u<grid.EndVertex()` and `u>=grid.StartVertex()` does not necessary mean that there is an edge in this grid starts from *u*. However, the generated lower-bound is always correct, because it is smaller than the actual lower-bound. Determining grid scheduling order based on the loose lower-bound is also correct, because the order only affect performance but not correctness.

4.3 Optimization

Abstraction The most effective optimization of Wonderland is abstraction-enabled information propagation: reusing an abstraction with X edges in each process iteration. For shortest path, graph abstraction is generated with a simple method: the negative of edge weight is used as the edge's priority so that X edges with smallest weights remain in abstraction. In Section 4.7, we will discuss the impact of X with evaluation results.

Upper-Bound Since we are processing shortest path queries rather than solving a single source shortest path problem, we can use the current *dist* property of the *dst* vertex as the upper-bound of relaxation. This optimization can be implemented in a straightforward manner by adding an *if* condition between line 8 and line 9 to check that whether `new_dist` is smaller than `g.Vertex(dst).dist`. Only if the condition is true, the following relaxation is performed.

Selective Loading There is no need to load a grid if all vertices between `grid.StartVertex()` and `grid.EndVertex()` are inactive. We integrate this optimization into our priority-based grid scheduling mechanism. We use a special priority "$-\infty$" to indicate that all the edges of a grid are not activated. For every loaded grid, its priority is firstly set to $-\infty$. Then, if some of its edges are activated, the grid's priority will be updated by line 25 in Figure 7. A grid is only loaded and processed when its priority is not $-\infty$. Moreover, with the upper bound optimization, we can avoid loading a grid if its estimated lower-bound is larger than the upper-bound, — current *dist* property of the *dst*.

4.4 Evaluation Environment

To evaluate the effectiveness of Wonderland and its optimizations, we use a machine 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 drive to perform the experiments. According to our evaluation, the average throughput of our SSD drive is about 760MB/s for sequential read. Moreover, in order to mimic the out-of-core setting and demonstrate the effect of reducing disk I/O, we use `cgroup` to set various *memory limits* (from 128MB to 32GB) for every query. Typically, the amount of disk I/O becomes larger when the memory limit is smaller.

4.5 Performance Comparison

In this section, we compare Wonderland with two other state-of-the-art systems (i.e., GridGraph [45] and Galois [29]). We choose GridGraph, because it is reported to be faster than many other out-of-core systems (e.g., X-Stream [30], GraphChi [21]). We consider an in-memory system Galois and its built-in out-of-core version (named LigraChi-g [29]) to show that our abstraction-based optimizations can even achieve considerable speedups in a full in-memory environment. It is worth noting that the three systems return the *same precise answer* (i.e., reach the same convergence condition), thus we can directly compare their performance.

The time needed for answering a shortest path query depends on input data (i.e., *src* and *dst*). We randomly select 30 pairs and report the average of their results. All experiments use 16 threads and all the reported execution times start from the point when the query is submitted. In other words, the preprocessing time (will be discussed later in Section 5.5), which can be amortized, is not included. We tested all possible parameter settings (e.g., the grid width in GridGraph, the delta of delta-Stepping algorithm in Galois) and only report the best results. The impact of Wonderland's parameters will be discussed in Section 4.7 and Section 4.9.

In this case study, we only report the results on LiveJournal [2] and Twitter [20], whose data size are about 790MB and 17GB respectively. Thus, the system is executed in out-of-core environment only when the memory limit is set to 512MB (for LiveJournal) and 16GB (for Twitter) or less, respectively. More datasets are used later in Section 5.4.

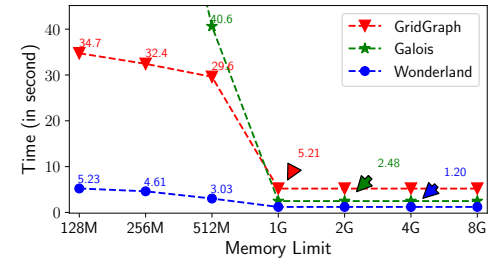
Fully In-Memory Setting As can see from Figure 8, in full-memory settings (when limit $\geq$ 1GB for LiveJournal and limit = 32GB for Twitter), the performance of Galois and Wonderland are much better than GridGraph. For example, when the limit is set to 32G, average execution time of GridGraph, Galois, and Wonderland are 81.4s, 45.1s, and 9.17s on Twitter, respectively. This aligns with the recent investigations [32], which show that a multi-thread Galois program's performance is even close to manually optimized native applications. However, Wonderland can still be 2.1× – 4.9× faster than Galois. This is not surprising as Wonderland is built on top of the executor similar to Galois's and, at the same time, takes advantage of an abstraction. Similar results have been reported in recent works that focus on using abstraction in full-memory environment [19].

Out-of-core Setting Different from the full-memory setting, the out-of-core version of Galois (i.e., LigraChi-g) is a simple synchronous executor and is much slower than GridGraph. For Twitter with 16G memory limit, Galois achieves better performance than GridGraph with the original `mmap`-based implementation (i.e., not LigraChi-g), because most data of Twitter (about 17G) can be cached in memory.

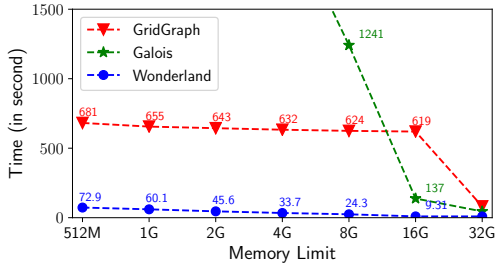
Since Wonderland can considerably accelerate the convergence speed, it is much faster (7.1× - 25.7×) than the better one of GridGraph and Galois. As we can see from Figure 8, there is a steep gap between GridGraph's performance on full-memory and out-of-core setting, even when the memory limit is enough for holding a large portion of the data (512MB for LiveJournal and 16GB for Twitter). This is because that in GridGraph (and most of the existing out-of-core systems), the input graph data is partitioned into disjoint parts and these parts are typically processed one by one in an iteration. Therefore, the first part is evicted from memory when the system processes the last. Several existing systems (include GridGraph) have proposed selectively scheduling

methods to skip unnecessary parts, but the improvement is limited. Although the performance of GridGraph does not decrease much when the memory limit is further reduced, it can be considered that GridGraph *cannot* take advantage of the extra memory, leading to the poor performance.

In contrast, Wonderland always caches a proper size of abstraction in memory and hence achieves better and stable performance. We show best performance among different choices of X. More detailed sensitivity study of abstraction size X on performance will be studied in Section 4.7. The main reason of Wonderland's speedup is due to the faster convergence speed and the corresponding reduced amount of disk I/O, — the direct benefits from abstraction. In order to validate our analysis, we check the I/O counters recorded in `/proc/{pid}/io`. Results show that the total read bytes caused by Wonderland can be *dozens of times less* than existing systems. To further understand this impact, in Section 4.6, we will present a piecewise breakdown on the effects of the three abstraction-based optimizations.



(a) LiveJournal.



(b) Twitter.

Figure 8. Performance of shortest path query.

4.6 Piecewise Breakdown

There are mainly three sources of speedup achieved by Wonderland: *1)* bootstrapping an initial result; *2)* abstraction-enabled information propagation; and *3)* abstraction-guided priority scheduling. In order to quantify the contribution of each source to the speedup, we perform a piecewise breakdown analysis by incrementally applying them. In Table 1, the numbers under "Init" column represent the performance when the abstraction is only used for generating an initial result. The "Bridge" column represent the performance when the both first two optimizations are applied. Finally, the "Priority" column are the final performance of Wonderland with all optimizations enabled.

From Table 1, we see that if the abstraction is only used for bootstrapping an initial result, its effect on Twitter dramatically decreases from 21× to only 2.2× when the memory limit is decreased from 16G to 4G. This is reasonable because, with less memory limit, less edges can remain in abstraction. With "Bridge" optimization, an extra speedup on top of "Init" is achieved in all cases (2×-4×). We also see that such effect could slightly increase when decreasing memory limit (from 2.2× to 3.5× for Twitter). Our priority-based loading mechanism is also effective as the final performance of Wonderland is even 1.6×-2.4× faster than "Bridge". In summary, if the memory is enough to accommodate a large abstract, the "Init" optimization contributes most for the speedups. In contrast, if the memory limit is low, the other two optimizations are necessary to achieve a good performance. Moreover, even when 16G memory is given for querying on a 17G dataset (i.e., Twitter), the two unique optimizations of Wonderland can still deliver a 3× speedup.

Table 1. Piecewise Breakdown Analysis.

Dataset	Limit	Average Execution Time (in second)			
		GridGraph	Wonderland		
			Init	Bridge	Priority
LiveJournal	512M	29.6	15.1	4.79	3.03
	256M	32.4	20.1	6.62	4.61
	128M	37.7	25.4	8.61	5.23
Twitter	16G	619	28.1	13.1	9.31
	8G	624	211	67.6	24.3
	4G	632	286	81.31	33.7

4.7 Abstraction Size Sensitivity

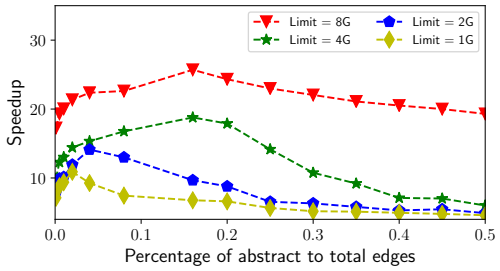


Figure 9. Impact of the Abstraction Size.

The size of abstraction (i.e., X) is a crucial parameter that affects the final performance. More interestingly, it incurs two opposite effects: *1)* the increase of X can reduce the number of iterations that are required to reach convergence; but *2)* since the abstraction is used to construct the in-memory sub-graph for every process iteration, the increase of X also increases the cost of each iteration.

Figure 9 shows the speedups at different ratios between abstraction size and total edge size (i.e., $X/|E|$) for different memory limits. While it is difficult to derive a formula that can automatically calculate the best X, the results show two patterns that help us to find the best X efficiently: *1)* for all memory size limits, the curve illustrating the relationship between speedup and $X/|E|$ is a unimodal function that first

increases and then decreases after a certain peak; and *2)* the position of the peak tends to be smaller if the memory limit is lower. Nevertheless, for all the four memory limits (1G-8G) and all the tested $X/|E| \in [0.001, 0.5]$, the achieved speedups are always more than 4.5×, — still considerable without making effort to choose the best X.

4.8 Multi-thread Speedup

The advantage of using multi-thread execution is usually not very significant for existing out-of-core graph processing systems, because the main bottleneck of their performance is disk I/O. For example, according to our evaluation, 16 threads only achieve less than 2.5× speedup over single thread in GridGraph. In comparison, Figure 10 presents the results of executing Wonderland with different number of threads. It clearly shows that the scalability of Wonderland is better than GridGraph. This is because that, for every iteration, not only the loaded edge grid but also the edges in abstraction need to be processed, leading to a better balance between CPU and disk I/O. Moreover, since Wonderland requires much less amount of disk I/O than GridGraph due to faster convergence, even without using multi-thread execution, the performance of single-thread Wonderland is better than 16-threads GridGraph.

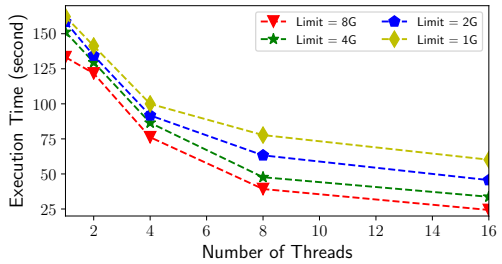


Figure 10. Scalability.

4.9 Grid Partition

The width of grid W and the maximum size of each grid S are two important parameters. We find that, as long as users do not choose extreme numbers (e.g., $A + S$ is too large such that the sub-graph cannot be fully held in memory), they do not present a significant impact on performance. Typically, setting S and W so that the size of each edge grid and vertex interval is about 32MB-64MB can lead to good-enough bandwidth utilization of the underlying SSD.

5 Other Applications

5.1 Applications

The programming model of Wonderland supports many kinds of graph applications that do not need to modify edge properties, i.e., the same scope as X-Stream, GridGraph, and out-of-core Galois. However, as Wonderland mainly takes advantage of graph abstraction, the speedup is related to certain property that is different from other systems.

For graph queries, the answer can be typically obtained by enumerating "all the paths" in the input graph. However, the number of paths grows exponentially with the size of graph. Fortunately, many advanced algorithms enable us to calculate the result of a given query by only checking a small subset of all the possible paths. For example, if we have already found a path with length l between u and v, we do not need to check any path longer than l to answer the shortest path query between u and v. We define the *selectivity* of a specific graph query as "(number of all the paths)/(number of paths that are needed to be checked)". The speedup achieved by Wonderland is typically positively related to the selectivity of the input graph query. With these definitions, we divide graph queries into two categories.

Any Path This kind of query can be answered by *any* path between two vertices. Querying the **reachability** between two vertices is a typical example. It is one of the simplest but important kind of query because it is used as a subroutine of many other graph algorithms. Moreover, finding weakly connected components (i.e., **WCC**) of a graph is another example of any-path query. Although it does not require two specific source and destination vertices, it can be considered as a combined any-path query on all vertex pairs.

All Path This kind of query can only be answered by checking *all* the paths between two vertices or all the vertices pairs. Although appearing exhaustive, effective pruning rules can be defined for many kind of queries in this category, which still ensure a high selectivity (e.g., **shortest path**). Another kind of all-path query is **widest path**. Specifically, the input of widest path query is the same as shortest path, i.e., a weighted graph, a source vertex and a destination vertex. But, rather than minimizing the aggregated weight of the path, widest path query tries to maximize the weight of the minimum-weight edge in the path. Similar to all the other three graph applications that we evaluated, widest path is another fundamental graph operation with wide application (e.g., finding bandwidth bottleneck of a network, calculating maximum flow, etc.).

Table 2. Real-World Graph Datasets.

Graph	Vertices	Edges	Data Size	Type	Diameter
LiveJournal [2]	4.85M	69.0M	790MB	Directed	15
Twitter [20]	41.7M	1.47B	17GB	Directed	13
Friendster [1]	65.6M	1.8B	41GB	Undirected	32
Dimacs [3]	23.9M	58.3M	668MB	Undirected	8122

5.2 Methodology

To evaluate the four graph applications discussed earlier, we use a collection of real-world graphs listed in Table 2. For undirected graphs, to test WCC, we double the edges to have them in both direction. For unweighted graphs, a random weight from (0,1) is attached to every edge for shortest and widest path. The environment is the same as the one described in Section 4.4. To ensure consistent comparison, for every dataset we only report the speedups achieved when the memory limit is set to 1/2, 1/4 and 1/8 of the input data

size. This makes the memory limit of different dataset different. We only report the speedup against GridGraph, because it is the fastest out-of-core system among all the four existing systems we test (i.e., GridGraph, X-Stream, GraphChi, and out-of-core Galois). The results are shown in Table 3.

5.3 Any-path Query

For reachability queries, due to the highest selectivity, Wonderland achieves the highest speedup on this kind of query (13.5×-376×). Even when only 1GB memory is given, Wonderland can still response to users within a second-level latency on a dataset that has 41GB (compared to more than 10 minutes for GridGraph). More importantly, this significant speedup is achieved without applying a large modification on the original BFS-based algorithm that GridGraph uses. Essentially, we simply *1) randomly* select a subset of the original graph's edge set as its abstraction; and *2)* set the priority of an activate edge grid according to a (similar) method described in Section 4.2.

For WCC, its selectivity is much lower because all the vertices pairs need to be checked. To achieve good speedup, we enable the connectivity optimization described in Section 3.2 which generates an abstraction that contains as few weakly connected components as possible. This optimization does not have a great impact for reachability queries because the original speedup is already very high. Overall, Wonderland outperforms GridGraph by 1.49×-9.03×[2]. Results also show that the speedup is higher if the input size or the diameter of the input graph is larger. According to our evaluation, only an average of 2× speedup is achieved if the WCC application uses a randomly selected abstraction. Thus, there exists a trade-off between query speedup and pre-processing cost, which will be further studied shortly in Section 5.5.

5.4 All-path Query

The implementation of shortest path query has been discussed intensively in Section 4. Thus, we simply present results from more datasets in Table 3, which show that Wonderland can also achieve a considerable speedup on graph that has larger size (Friendster) or larger diameter (Dimacs).

The implementation of widest path query is very similar to the shortest path query. The only differences are: *1)* edges with largest rather than smallest weights remain in the abstraction; and *2)* the priority of an edge grid is decided by the maximum-weight edge this grid contains. As we can see from Table 3, the speedups of widest path typically share the same trend of shortest path queries. The only exception is on Dimacs, where the speedup largely increases to 41.3× after raising the memory limit to 1/4 of the data size. This is because, rather than using the randomly generated weights, Dimacs uses the real-world weights, which are more skewed and hence most of the high-weight edges remain in the abstraction.

[2]In fact, we also evaluated the connectivity query, which is a single-pair version of WCC query that simply answers whether two vertices are connected by an undirected path. The speedup is even higher than reachability queries even with randomly-selected abstraction.

Table 3. Speedup on different datasets and memory limits.

Memory Limit		1/2	1/4	1/8
Reachbility	LiveJournal	19.8×	16.1×	13.5×
	Twitter	86.6×	67.0×	45.7×
	Frindster	376×	247×	201×
	Dimacs	55.3×	39.1×	10.7×
WCC	LiveJournal	4.85×	3.75×	1.82×
	Twitter	6.18×	5.90×	3.78×
	Frindster	7.11×	6.20×	4.89×
	Dimacs	9.03×	4.32×	1.49×
Shortest Path	LiveJournal	9.92×	8.05×	3.53×
	Twitter	24.7×	20.9×	15.4×
	Frindster	36.2×	24.2×	17.3×
	Dimacs	17.8×	5.91×	2.13×
Widest Path	LiveJournal	11.3×	6.18×	3.03×
	Twitter	9.68×	6.91×	6.04×
	Frindster	20.1×	17.5×	11.2×
	Dimacs	63.2×	41.3×	2.65×

5.5 Preprocessing Time

Table 4 presents the pre-processing cost of Wonderland. In this table, *1)* column "Random" means that the abstraction is a random subset of the original graph's edge set, which is enough for reachability and connectivity queries; *2)* column "Order" means that an edge priority function defined by users is used in the abstraction generating phase, whose results can be used for shortest or widest path querying; and *3)* column "Connectivity" means that the connectivity optimization described in Section 3.2 is further enabled.

Table 4. Preprocessing Cost (in seconds).

Dataset	GridGraph	Wonderland		
		Random	Order	Connectivity
LiveJournal	5.53	7.09	9.15	18.6
Twitter	92.8	101	161	317
Frindster	235	295	398	912
Dimacs	3.79	6.29	6.35	11.6

As we can see from the table, the preprocessing cost of Wonderland is longer but less than 2× of GridGraph without the connectivity information. Since the significant speedup for the following queries could amortize the slightly longer preprocessing time, the overall execution time of Wonderland is still smaller than GridGraph even when *only one* query is submitted. In fact, the preprocessing cost can be further amortized by reusing its results. Practically, it is also possible to merge the preprocessing procedure of multiple kinds of queries by generating multiple abstractions at the same time. In our experiments, only about 200s are needed to accomplish the preprocessing cost of reachability, connectivity, shortest path, and widest path simultaneously for the Twitter dataset.

In contrast, the overhead of connectivity optimization is relatively high. It is mainly because that our current implementation simply uses a hash-map-based disjoint-set method, which does not support concurrent updates. However, since we actually do not need to precisely maintaining the connectivity information, it is possible to reduce this cost by approximation and parallelization, which we leave as the future work.

5.6 Scope of Application

As described in Section 5.1, Wonderland leads to higher speedup if the original problem has a higher "selectivity". Thus, Wonderland can deliver much better performance for applications that analysis the graph structures, including the applications listed above and others like (BFS, MST). Moreover, all the listed applications are basic graph operations that prevalently used to construct more complex applications like clustering [33], matching [40], maximum flow [11], community or anomaly detection [5, 9], which can all benefit from the proposed technique. The strong evidence suggests that Wonderland is applicable for many important kinds of graph applications. In the other side, according to our evaluation, although Wonderland can also be faster than GridGraph in computing SpMV-based algorithms (e.g., PageRank), it is due to the Galois-like execution engine, but not the abstraction.

Note that it is difficult to define a precise scope of algorithms can be optimized by Wonderland, because different problems require different abstractions to achieve the best performance. To enhance programmability, we provide simple APIs for facilitating users to select the abstractions, rather than design a specific abstraction-generating algorithm like Kusum *et al.* [19]. As described in Section 3.2, using our APIs, abstraction generation only requires a few line of codes. Our evaluation results firmly validate that these simple abstractions are sufficient to gain significant speedups.

6 Related Works

Existing works related to out-of-core graph processing and graph abstraction are discussed in Section 2.1 and Section 2.2, respectively. Our work is different from them in a number of aspects: *1).* To generate abstraction, existing methods [14, 15, 17, 19, 22, 25, 27, 28, 31, 43] usually require complex graph transformations that can only be implemented efficiently in a full memory environment. Thus, they are not applicable in an out-of-core environment. *2).* Existing abstraction-based methods only use abstraction to generate an initial result. We propose two novel techniques based abstractions, which are unique in our work. More importantly, we demonstrate that each of the three usages deliveries considerable speedup. *3).* While some existing works [19] use abstraction to generate approximate results, Wonderland always returns precise answers (i.e., reach to the same convergence point as Galois/GridGraph).

Some other works, such as CLIP [4] also tries to reduce the number of iterations and the amount of total disk I/O, but it depends on a programming model that allows the random access of all vertices (but not all edges). Wonderland does not have this requirement and is based on the existing vertex programing model. Essentially, CLIP accelerates convergence by better algorithms (with random vertex accesses), Wonderland achieves the same with abstraction. Another work, GraphQ [39], also uses abstraction to acclerate graph query. However, GraphQ only accelrates analytical queries with the form of "find n entities from the graph with a given quantitative property". These queries by nature can be answered with local information. In contrast, Wonderland is used for traditional queries that require precise answer with global information. When GraphQ is used for the traditional queries (i.e., without a limitation), due to whole-graph computation requirement and the lack of an effective merging mechanism, eventually all partitions will be merged into memory. If a machine does not have enough memory to store the whole graph, then GraphQ cannot handle the query.

Besides these works, there are also many single-machine and distributed computing systems that perform graph processing in memory [12, 13, 24, 26, 34, 41, 44]. These works are orthogonal to ours because we focus on problems specific to the out-of-core environment.

Some existing works [21, 36] are proposed to support evolving graphs, which is also orthogonal to this paper. While not discussed, some of these techniques can be integrated into Wonderland easily, e.g., the method used for handling adding edges in GraphChi [21]. Simply caching all the recently added edges in the abstraction and periodically invoking the `Select` function is also a feasible method. As for deleting edges, our system can support this functionality by attaching a label to each edge to determine whether it still exists in the graph. Some works [37] try to enable incremental processing after deleting edges, we will incorporate abstraction in this scenario in future work. However, since Wonderland's querying performance is already much higher than existing systems, we believe a simple re-execution may be acceptable in many cases.

7 Conclusion

This paper proposes Wonderland, a novel out-of-core graph processing system based on abstraction with three unique features: *1)* A simple method is proposed to allow users to extract effective abstractions from the original graph with acceptable cost and a specific memory limit; *2)* Abstraction-enabled information propagation, where an abstraction can be used as a bridge over the disjoint on-disk graph partitions; *3)* Abstraction-guided priority scheduling, where an abstraction can infer the better priority-based order in processing on-disk graph partitions. Wonderland is a significant advance over the state-of-the-art because it not only makes graph abstraction feasible to out-of-core systems, but also

broadens the applications of the concept in important ways. Evaluation results on Wonderland reveal that Wonderland achieves a drastic speedup over the other state-of-the-art systems, — up to two orders of magnitude for certain cases.

Acknowledgments

This work is supported by the following grants: National Key Research & Development Program of China (2016YFB1000504), Natural Science Foundation of China (61433008, 61373145, 61572280, U1435216, 61402198), National Basic Research (973) Program of China (2014CB340402), NSF-CCF-1657333, NSF-CCF-1717754, NSF-CNS-1717984, and NSF-CCF-1750656. Contact: Yongwei Wu (wuyw@tsinghua.edu.cn), Kang Chen (chenkang@tsinghua.edu.cn).

References

[1] [n. d.]. S. N. A. Project. Stanford large network dataset collection. http://snap.stanford.edu/data/com-Friendster.html. ([n. d.]).

[2] [n. d.]. S. N. A. Project. Stanford large network dataset collection. http://snap.stanford.edu/data/soc-LiveJournal1.html. ([n. d.]).

[3] [n. d.]. The Center for Discrete Mathematics and Theoretical Computer Science. http://www.dis.uniroma1.it /challenge9/download.shtml. ([n. d.]).

[4] Zhiyuan Ai, Mingxing Zhang, Yongwei Wu, Xuehai Qian, Kang Chen, and Weimin Zheng. 2017. 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 ATC 17)*. USENIX Association, Santa Clara, CA, 125–137.

[5] Leman Akoglu, Hanghang Tong, and Danai Koutra. 2015. Graph Based Anomaly Detection and Description: A Survey. *Data Min. Knowl. Discov.* 29, 3 (May 2015), 626–688. https://doi.org/10.1007/s10618-014-0365-y

[6] David A. Bader, Shiva Kintali, Kamesh Madduri, and Milena Mihail. 2007. Approximating Betweenness Centrality. In *Proceedings of the 5th International Conference on Algorithms and Models for the Web-graph (WAW'07)*. Springer-Verlag, Berlin, Heidelberg, 124–137.

[7] Bernard Chazelle, Ronitt Rubinfeld, and Luca Trevisan. 2001. Approximating the Minimum Spanning Tree Weight in Sublinear Time. In *Proceedings of the 28th International Colloquium on Automata, Languages and Programming, (ICALP '01)*. Springer-Verlag, London, UK, UK, 190–200.

[8] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. 2001. *Introduction to Algorithms* (2nd ed.). McGraw-Hill Higher Education.

[9] Santo Fortunato. 2010. Community detection in graphs. *Physics reports* 486, 3 (2010), 75–174.

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

[11] Andrew V. Goldberg and Robert E. Tarjan. 2014. Efficient Maximum Flow Algorithms. *Commun. ACM* 57, 8 (Aug. 2014), 82–89. https://doi.org/10.1145/2628036

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

[13] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. 2014. Graphx: Graph processing in a distributed dataflow framework. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 599–613.

[14] Bruce Hendrickson and Robert Leland. 1995. A Multilevel Algorithm for Partitioning Graphs. In *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing (Supercomputing '95)*. ACM, New York, NY, USA, Article 28. https://doi.org/10.1145/224170.224228

[15] George Karypis and Vipin Kumar. 1998. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.* 20, 1 (Dec. 1998), 359–392. https://doi.org/10.1137/S1064827595287997

[16] George Karypis and Vipin Kumar. 1998. Multilevelk-way Partitioning Scheme for Irregular Graphs. *J. Parallel Distrib. Comput.* 48, 1 (Jan. 1998), 96–129. https://doi.org/10.1006/jpdc.1997.1404

[17] A. A. Khan, M. U. Khan, and M. Iqbal. 2012. Multilevel Graph Partitioning Scheme to Solve Traveling Salesman Problem. In *2012 Ninth International Conference on Information Technology - New Generations*. 458–463. https://doi.org/10.1109/ITNG.2012.106

[18] Pradeep Kumar and H. Howie Huang. 2016. G-store: High-performance Graph Store for Trillion-edge Processing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '16)*. IEEE Press, Piscataway, NJ, USA, Article 71, 12 pages.

[19] Amlan Kusum, Keval Vora, Rajiv Gupta, and Iulian Neamtiu. 2016. Efficient Processing of Large Graphs via Input Reduction. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC '16)*. ACM, New York, NY, USA, 245–257. https://doi.org/10.1145/2907294.2907312

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

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

[22] Kristen LeFevre and Evimaria Terzi. 2010. GraSS: Graph structure summarization. In *Proceedings of the 2010 SIAM International Conference on Data Mining*. SIAM, 454–465.

[23] Hang Liu and H. Howie Huang. 2017. Graphene: Fine-Grained IO Management for Graph Computing. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. USENIX Association, Santa Clara, CA, 285–300.

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

[25] M. Riondato and D. Garcia-Soriano and F. Bonchi. 2014. Graph Summarization with Quality Guarantees. In *2014 IEEE International Conference on Data Mining*. 947–952. https://doi.org/10.1109/ICDM.2014.56

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

[27] Irene Moulitsas and George Karypis. 2001. Multilevel Algorithms for Generating Coarse Grids for Multigrid Methods. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing (SC '01)*. ACM, New York, NY, USA, 45–45. https://doi.org/10.1145/582034.582079

[28] Danupon Nanongkai. 2014. Distributed Approximation Algorithms for Weighted Shortest Paths. In *Proceedings of the Forty-sixth Annual ACM Symposium on Theory of Computing (STOC '14)*. ACM, New York, NY, USA, 565–573. https://doi.org/10.1145/2591796.2591850

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

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

[31] Ning Ruan, Ruoming Jin, and Yan Huang. 2011. Distance Preserving Graph Simplification. In *Proceedings of the 2011 IEEE 11th International Conference on Data Mining (ICDM '11)*. IEEE Computer Society, Washington, DC, USA, 1200–1205. https://doi.org/10.1109/ICDM.2011.57

[32] Nadathur Satish, Narayanan Sundaram, Md. Mostofa Ali Patwary, Jiwon Seo, Jongsoo Park, M. Amber Hassaan, Shubho Sengupta, Zhaoming Yin, and Pradeep Dubey. 2014. Navigating the Maze of Graph Analytics Frameworks Using Massive Graph Datasets. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*. ACM, New York, NY, USA, 979–990. https://doi.org/10.1145/2588555.2610518

[33] Satu Elisa Schaeffer. 2007. Survey: Graph Clustering. *Comput. Sci. Rev.* 1, 1 (Aug. 2007), 27–64. https://doi.org/10.1016/j.cosrev.2007.05.001

[34] Jiaxin Shi, Youyang Yao, Rong Chen, Haibo Chen, and Feifei Li. 2016. Fast and Concurrent RDF Queries with RDMA-Based Distributed Graph Exploration. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, GA, 317–332.

[35] Yuanyuan Tian, Richard A. Hankins, and Jignesh M. Patel. 2008. Efficient Aggregation for Graph Summarization. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD '08)*. ACM, New York, NY, USA, 567–580. https://doi.org/10.1145/1376616.1376675

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

[37] Keval Vora, Rajiv Gupta, and Guoqing Xu. 2017. Kickstarter: Fast and accurate computations on streaming graphs via trimmed approximations. ASPLOS.

[38] Keval Vora, Guoqing Xu, and Rajiv Gupta. 2016. 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.

[39] Kai Wang, Guoqing Xu, Zhendong Su, and Yu David Liu. 2015. GraphQ: Graph Query Processing with Abstraction Refinement: Scalable and Programmable Analytics over Very Large Graphs on a Single PC. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '15)*. USENIX Association, Berkeley, CA, USA, 387–401.

[40] Junchi Yan, Xu-Cheng Yin, Weiyao Lin, Cheng Deng, Hongyuan Zha, and Xiaokang Yang. 2016. A Short Survey of Recent Advances in Graph Matching. In *Proceedings of the 2016 ACM on International Conference on Multimedia Retrieval (ICMR '16)*. ACM, New York, NY, USA, 167–174. https://doi.org/10.1145/2911996.2912035

[41] Mingxing Zhang, Yongwei Wu, Kang Chen, Xuehai Qian, Xue Li, and Weimin Zheng. 2016. Exploring the Hidden Dimension in Graph Processing. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, GA, 285–300.

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

[43] Fang Zhou, Sebastien Malher, and Hannu Toivonen. 2010. Network Simplification with Minimal Loss of Connectivity. In *Proceedings of the 2010 IEEE International Conference on Data Mining (ICDM '10)*. IEEE Computer Society, Washington, DC, USA, 659–668. https://doi.org/10.1109/ICDM.2010.133

[44] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A Computation-Centric Distributed Graph Processing System. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, GA, 301–316.

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