Graphene: Fine-Grained IO Management for Graph Computing

Hang Liu and H. Howie Huang, ***The George Washington University***

https://www.usenix.org/conference/fast17/technical-sessions/presentation/liu

This paper is included in the Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST '17).

February 27–March 2, 2017 • Santa Clara, CA, USA

ISBN 978-1-931971-36-2

Open access to the Proceedings of the 15th USENIX Conference on File and Storage Technologies is sponsored by USENIX.

Graphene: Fine-Grained IO Management for Graph Computing

Hang Liu H. Howie Huang
The George Washington University
{asherliu, howie}@gwu.edu

Abstract

As graphs continue to grow, external memory graph processing systems serve as a promising alternative to in-memory solutions for low cost and high scalability. Unfortunately, not only does this approach require considerable efforts in programming and IO management, but its performance also lags behind, in some cases by an order of magnitude. In this work, we strive to achieve an ambitious goal of achieving ease of programming and high IO performance (as in-memory processing) while maintaining graph data on disks (as external memory processing). To this end, we have designed and developed *Graphene* that consists of four new techniques: an IO request centric programming model, bitmap based asynchronous IO, direct hugepage support, and data and workload balancing. The evaluation shows that Graphene can not only run several times faster than several external-memory processing systems, but also performs comparably with in-memory processing on large graphs.

1 Introduction

Graphs are powerful data structures that have been used broadly to represent the relationships among various entities (e.g., people, computers, and neurons). Analyzing massive graph data and extracting valuable information is of paramount value in social, biological, healthcare, information and cyber-physical systems [14, 15, 17, 24, 29].

Generally speaking, graph algorithms include reading the *graph data* that consists of a list of neighbors or edges, performing calculations on vertices and edges, and updating the *graph (algorithmic) metadata* that represents the states of vertices and/or edges during graph processing. For example, breadth-first search (BFS) needs to access the adjacency lists (data) of the vertices that have just been visited at the prior level, and mark the statuses (metadata) of previously unvisited neighbors as visited. Accesses of graph data and metadata come hand-in-hand in many algorithms, that is, reading one vertex or edge will be accompanied with access to the corresponding metadata. It is important to note that in this paper we use the term *metadata* to refer to the key data structures in graph computing (e.g., the statuses in BFS and the ranks in PageRank).

To tackle the IO challenge in graph analytics, prior research utilizes in-memory processing that stores the whole graph data and metadata in DRAM to shorten the latency of random accesses [20, 35, 40, 44, 47]. In-memory processing brings a number of benefits including easy programming and high-performance IOs. However, this approach is costly and difficult to scale, as big graphs continue to grow drastically in size. On the other hand, the alternative approach of external memory graph processing focuses on accelerating data access on storage devices. However, this approach suffers not only from complexity in programming and IO management but also slow IO and overall system performance [40, 62].

To close the gap between in-memory and external memory graph processing, we design and develop *Graphene*, a new semi-external memory processing system that efficiently reads the graph data on SSDs while managing the metadata in DRAM. Simply put, Graphene incorporates graph data awareness in IO management behind an IO centric programming model, and performs fine-grained IOs on flash-based storage devices. This is different from current practice of issuing large IOs and relying on operating system (OS) for optimization [40, 47, 62]. Figure 1 presents the system architecture. The main contributions of Graphene are four-fold:

IO (request) centric graph processing. Graphene advocates a new paradigm where each step of graph processing works on the data returned from an IO request. This approach is unique from four types of existing graph processing systems: (1) vertex-centric programming model, e.g., Pregel [36], GraphLab [35], PowerGraph [20], and Ligra [47]; (2) edge-centric, e.g., X-stream [44] and Chaos [43]; (3) embedding-centric, e.g., Arabesque [50]; and (4) domain-specific language, e.g.,

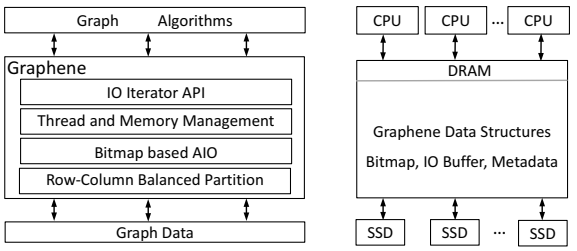


Figure 1: Architecture overview.

Galois [40], Green-Marl [27] and Trinity [46]. All these models are designed to address the complexity of the computation, including multi-threaded processing [27, 40], workload balancing [10, 20], inter-thread (node) communication [38] and synchronization [36]. However, in order to achieve good IO performance, these models require a user to explicitly manage the IOs, which is a challenging job by itself. For example, FlashGraph needs user input to sort, merge, submit and poll IO requests [62].

In Graphene, IO request centric processing (or IO centric for short) aims to simplify not only graph programming but also the task of IO management. To this end, we design a new *IoIterator* API that consists of a number of system and user-defined functions. As a result, various graph algorithms can be written in about 200 lines of code. Behind the scenes, Graphene translates high-level data accesses to fine-grained IO requests for better optimization. In short, IO centric processing is able to retain the benefit of easy programming while delivering high-performance IO.

Bitmap based, asynchronous IO. Prior research aims to read a large amount of graph data as quickly as possible, even when only a portion of it is needed. This design is justified because small random accesses in graph algorithms are not the strong suit of rotational hard drives. Notable examples include GraphChi [32] and X-stream [44], which read the entire graph data sequentially from the beginning to the end during each iteration of the graph calculation. In this case, the pursuit of high IO bandwidth overshadows the usefulness of data accesses. Besides this full IO model, the IO on-demand approach loads only the required data in memory, but again requires significant programming effort [25, 56, 62].

With the help of IO centric processing, Graphene pushes the envelope of the IO on-demand approach. Specifically, Graphene views graph data files as an array of 512-byte blocks, a finer granularity than more commonly used 4KB, and uses a *Bitmap*-based approach to quickly reorder, deduplicate, and merge the requests. While it incurs 3.4% overhead, the Bitmap approach improves the *IO utility* by as much as 50%, and as a result runs more than four times faster than a typical list based IO. In this work, IO utility is defined as the ratio between the amount of data that is loaded and useful for graph computation, and that of all the data loaded from disk. Furthermore, Graphene exploits *Asynchronous IO* (AIO) to submit as many IO requests as possible to saturate the IO bandwidth of flash devices.

Direct hugepage support. Instead of using 4KB memory pages, Graphene leverages the support of *Direct HugePage* (DHP), which preallocates the (2MB and 1GB) hugepages at boot time and uses them for both graph data and metadata structures, e.g., IO buffer and Bitmap. For example, Graphene designs a hugepage based memory buffer which enables multiple IO requests to share one hugepage. This technique eliminates the runtime uncertainty and high overhead in the transparent hugepage (THP) method [39], and significantly lowers the TLB miss ratio by 177×, leading to, on average, 12% performance improvement across different algorithms and graph datasets.

Balanced data and workload partition. Compared to existing 2D partitioning methods which divide vertices into equal ranges, Graphene introduces a row-column balanced 2D partitioning where each partition contains an equal number of edges. This ensures that each SSD holds a balanced data partition, especially in the cases of highly skewed degree distribution in real-world graphs. However, a balanced data partition does not guarantee that the workload from graph processing is balanced. In fact, the computation performed on each partition can vary drastically depending on the specific algorithm. To address this problem, Graphene utilizes dedicated IO and computing threads per SSD and applies a work stealing technique to mitigate the imbalance within the system.

We have implemented Graphene with different graph algorithms and evaluated its performance on a number of real world and synthetic graphs on up to 16 SSDs. Our experiments show that Graphene outperforms several external memory graph systems by 4.3 to 20×. Furthermore, Graphene is able to achieve similar performance to in-memory processing with the exception of BFS.

This paper is organized as follows: Section 2 presents the IO centric programming model. Section 3 discusses bitmap-based, asynchronous IO and Section 4 presents data and workload balancing techniques, and Section 5 describes hugepage support. Section 6 describes a number of graph algorithms used in this work. Section 7 presents the experimental setup and results. Section 8 discusses the related work and Section 9 concludes.

2 IO Request Centric Graph Processing

Graphene allows the system to focus on the data, be it a vertex, edge or subgraph, returned from an IO request at a time. This new IO (request) centric processing aims to provide the illusion that all graph data resides in mem-

Table 1: IoIterator API

Type	Name	Return Value	Description
System provided	Iterator->Next()	io_block_t	Get the next in-memory data block
	Iterator->HasMore()	bool	Check if there are more vertices available from IO
	Iterator->Current()	vertex	Get the next available vertex *v*
	Iterator->GetNeighbors(vertex *v*)	vertex array	Get the neighbors for the vertex *v*
User defined	IsActive(vertex *v*)	bool	Check if the vertex *v* is active
	Compute(vertex *v*)		Perform algorithm specific computation

```
while true do
    foreach vertex v do
        if IsActive(v) then
            handle = IO_Submit(v);
            IO_Poll(handle);
            Compute(the neighbors of v);
        end
    end
    level++;
end
```

Algorithm 1: BFS with user-managed IO.

```
while true do
    block = IoIterator→Next();
    while block→HasMore() do
        vertex v = block→Current();
        if IsActive(v) then
            Compute(block→GetNeighbors(v));
        end
    end
    level++;
end
```

Algorithm 2: IoIterator-based BFS.

ory, and delivers high IO performance through applying various techniques behind the scenes which will be described in next three sections.

To this end, Graphene develops an *IoIterator* framework, where a user only needs to call a simple *Next()* function to retrieve the needed graph data for processing. This allows the programmers to focus on graph algorithms without worrying about the IO complexity in semi-external graph processing. At the same time, by taking care of graph IOs, the IoIterator framework allows Graphene to perform disk IOs more efficiently in the background and make them more cache friendly. It is worth noting that the IO centric model can be easily integrated with other graph processing paradigms including vertex or edge centric processing. For example, Graphene has a user-defined *Compute* function that works on vertices.

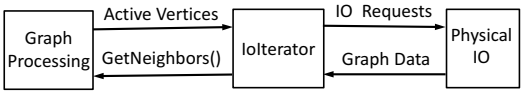


Figure 2: IoIterator programming model.

At a high level shown in Figure 2, we insert a new IoIterator layer between the algorithm and physical IO. In this architecture, the processing layer is responsible for the control flow, e.g., computing what vertices of the graph should be active, and working on the neighbors of those active vertices. The IO layer is responsible for serving the IO requests from storage devices. Graph processing can start as soon as the IOs for the adjacency lists of the active vertices are complete, i.e., when the data for the neighbors become available. The new abstraction of IoIterator is responsible for translating the requests for the adjacency lists into the IO requests for data blocks.

Internally, Graphene applied a number of IO optimizations behind the IoIterator, including utilizing a Bitmap per device for sorting and merging, submitting large amounts of non-blocking requests via asynchronous IO, using hugepages to store graph data and metadata, and resolving the mismatch between IO and processing across devices.

The IoIterator layer consists of a set of APIs listed in Table 1. There are four system-defined functions for the IoIterator, *Next*, *HasMore*, *Current*, and *GetNeighbors*, which work on the list of the vertices returned from the underlying IO layer. In addition, two functions *IsActive* and *Compute* should be defined by the users. For example, in BFS, the IsActive function should return *true* for any frontier if a vertex *v* has been visited in the preceding iteration, and Compute should check the status of each neighbor of *v*, and mark any unvisited neighbors as frontiers for the next iteration. Detailed description of BFS and other algorithms can be found in Section 6.

An example of BFS pseudocode written with the current approach of user-managed selective IO vs. the IoIterator API can be found in Algorithms 1 and 2. In the first approach, the users are required to be familiar with the Linux IO stack and explicitly manage the IO requests such as IO submission, polling, and exception handling. The main advantage of the IoIterator is that it completely removes such a need. On the other hand, in both approaches, the users need to provide two similar functions, *IsActive* and *Compute*.

It is important to note that the pseudocode will largely stay the same for other algorithms, but with different *IsActive* and *Compute*. For example, in PageRank, IsActive returns *true* for vertices that have delta updates, and Compute accumulates the updates from different source vertices to the same destination vertex. Here, Compute may be written in vertex or edge centric model.

3 Bitmap Based, Asynchronous IO

Graphene achieves high-performance IO for graph processing through a combination of techniques including fine-grained IO blocks, bitmap, and asynchronous IO. Specifically, Graphene favors small, 512-byte IO blocks to minimize the alignment cost and improve the IO utility, and utilizes a fast bitmap-based method to reorder and produce larger IO requests, which will be submitted to devices asynchronously. As a result, the performance of graph processing improves as a higher fraction of useful data are delivered to CPUs at high speed.

In Graphene, graph data are stored on SSDs in *Compressed Sparse Row* (CSR) format which consists of two data structures: the *adjacency list array* that stores the IDs of the destination vertices of all the edges ordered by the IDs of the source vertices, and the *beginning position array* that maintains the index of the first edge for each vertex.

3.1 Block Size

One trend in modern operating systems is to issue IOs in larger sizes, e.g., 4KB by default in some Linux distributions [8]. While this approach is used to achieve high sequential bandwidth from underlying storage devices like hard drives, doing so as in prior work [62] would lead to low IO utility because graph algorithms inherently issue small data requests. In this work, we have studied the IO request size when running graph algorithms on Twitter [2] and Friendster [1]. Various graph datasets that are used in this paper is summarized in Section 7. One can see that most (99%) of IO requests are much smaller than 4KB as shown in Figure 3. Thus, issuing 4KB IOs would waste a significant amount of IO bandwidth.

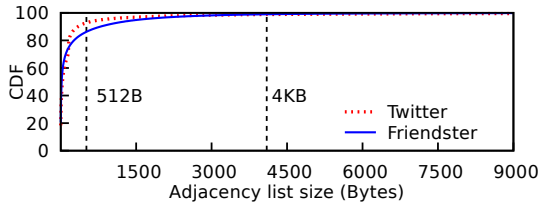


Figure 3: Distribution of IO sizes.

In Graphene, we choose to use a small IO size of 512 bytes as the basic block for graph data IOs. Fortunately, new SSDs are capable of delivering good IOPS for 512-byte read requests for both random and sequential IOs. For example, Samsung 850 SSD [49], which we use in the experiments, can achieve more than 20,000 IOPS for 512-byte random read.

Another benefit of using 512-byte blocks is to lower the cost of the alignment for multiple requests. Larger block size like 4KB means the offset and size of each IO request should be a multiple of 4KB. In the example presented in Figure 4, requesting the same amount of

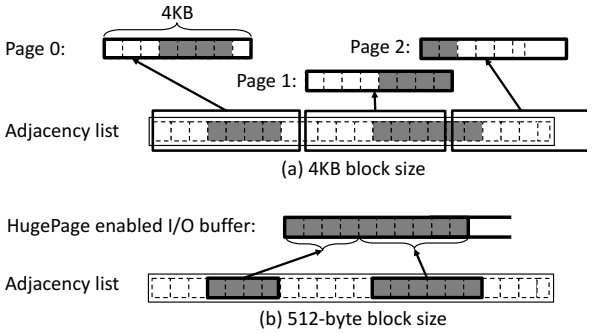


Figure 4: IO alignment cost: 4KB vs. 512-byte blocks, where one dotted box represents one 512-byte block.

data will lead to the different numbers of IOs when using 4KB (top) and 512-byte (bottom) block sizes. One can see that the former will load 2.2× more data, i.e., 12KB vs. 5KB in this case. In addition, combined with hugepage support that will be presented shortly, 512-byte block IO will need only one hugepage-based IO buffer, compared to three 4KB pages required in the top case.

3.2 Bitmap-Based IO Management

At each iteration of graph processing, graph algorithms compute and generate the requests for the adjacency lists (i.e., the neighboring vertices) of *all* active vertices for the following iteration. In particular, Graphene translates such requests into a number of 512-byte aligned IO blocks, which are quickly identified in a new *Bitmap* data structure. In other words, Graphene maintains a Bitmap per SSD, one bit for each 512-byte block on the disk. For each request, Graphene marks the bits for the corresponding blocks, that is, should a block need to be loaded, its bit is marked as "1", and "0" otherwise. Clearly, the Bitmap offers a *global* view of IO operations and enables optimization opportunities which would not otherwise be possible.

For a 500GB SSD as we have used in this work, the size of the bitmap is merely around 128MB, which we can easily cache in CPUs and store in DRAM with a number of hugegages. Because Graphene combines Bitmap-based management with asynchronous IO, it is also able to utilize one IO thread per SSD. Therefore, since there is only one thread managing the Bitmap for each SSD, no lock is required on the Bitmap structures.

Issues with local IO optimization. Traditionally, the OS takes a *local* view of the IO requests by immediately issuing the requests for the neighbors of one or a group of active vertices. In addition, the OS performs several important tasks such as IO batching, reordering and merging at the block layer. Unfortunately, these techniques have been applied only to IO requests that have been buffered in certain data structures. For instance, Linux exploits a linked list called *pluglist* to batch and submit the IO requests [8], in particular, the most recent Linux kernel 4.4.0 supports 16 requests in a batch.

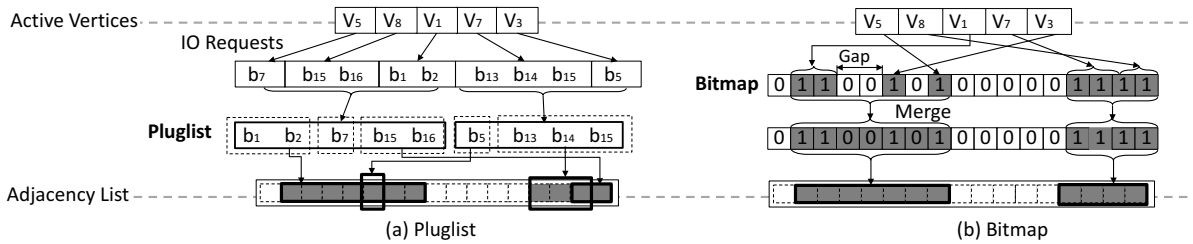


Figure 5: Pluglist vs. bitmap IO management, (a) Pluglist where sorting and merging are limited to IO requests in the pluglist. (b) Bitmap where sorting and merging are applied to all IO requests.

Figure 5(a) presents the limitations of the pluglist based approach. In this example, vertices $\{v_5, v_8, v_1, v_7, v_3\}$ are all active and the algorithm needs to load their neighbors from the adjacency list file. With a fixed-size pluglist, some of the requests will be *batched* and enqueued first, e.g., the requests for the first three vertices $\{v_5, v_8, v_1\}$. In the second step, *sorting* is applied across the IO requests in the pluglist. Since the requests are already grouped, sorting happens within the boundary of each group. In this case, the requests for the first three vertices are reordered from $\{b_7, b_{15}, b_{16}, b_1, b_2\}$ to $\{b_1, b_2, b_7, b_{15}, b_{16}\}$. In the third step, if some IO blocks present good spatial locality, *merging* will be applied to form a larger IO request, e.g., blocks $\{b_1, b_2, b_7\}$ are merged into one IO transaction. And later, a similar process happens for the IOs on the rest of vertices $\{v_7, v_3\}$.

In this case, there are four independent IO requests to the disk, (a) blocks b_1 - b_7, (b) blocks b_{15} - b_{16}, (c) block b_5, and (d) blocks b_{13} - b_{15}. The first request loads seven sequential blocks in one batch, which takes advantage of prefetching and caching and is preferred by the disks and OS. As a result, the third request for block b_5 will likely hit in the cache. On the other hand, although the second and fourth requests have overlapping blocks, they will be handled as two separate IO requests.

Bitmap and global IO optimization. Graphene chooses to carry out IO management optimizations, including IO deduplication, sorting and merging, on a *global* scale. This is motivated by the observation that although graph algorithms tend to present little or no locality in a short time period, there still exists a good amount of locality within the entire processing window. Bitmap-based IO management is shown in Figure 5(b). Upon receiving the requests for all active vertices, Graphene will convert the needed adjacency lists into the block addresses and mark those blocks in the Bitmap.

Sorting. The process of marking active blocks in the corresponding locations in the Bitmap naturally sorts the requests in the order of physical addresses on disks. In other words, the order of the requests is simply that of the marked bits in the Bitmap.

IO deduplication is also easily achieved in the process. Bitmap-based IO ensures that only one IO request will be sent even when the data block is requested multiple times, achieving the effect of IO deduplication. This is common in graph computation. For example, in the single source shortest path algorithm, one vertex may have many neighboring vertices, and if more than one neighbors need to update the distance of this vertex, it will need to be enqueued multiple times for the next iteration. In addition, different parts of the same IO block may need to be loaded at the same time. In the prior example, as the block b_{15} is shared by the requests from vertices v_7 and v_8, it will be marked and loaded once. Our study shows that the deduplication enabled from Bitmap can save up to 3× IO requests for BFS, compared to a pluglist based method.

IO merging. Bitmap is very easy to use for merging the requests in the vicinity of each other into a larger request, which reduces the total number of IO requests submitted to disks. For example, as shown in Figure 5(b), IO requests for vertices v_1, v_3, v_5 (and similarly for vertices v_7 and v_8) are merged into one. As a result, there are only two non-overlapping requests instead of four as in the pluglist case.

How to merge IO requests is guided by a number of rules. It is straightforward that consecutive requests should be merged. When there are multiple non-consecutive requests, we can merge them when the blocks to be loaded are within a pre-defined *maximum gap*, which determines the largest distance between two requests. Note that this rule directly evaluates the Bitmap by bytes to determine whether eight consecutive blocks are needed to be merged.

This approach favors larger IO sizes and has proven to be effective in achieving high IO performance. Figure 6 shows the performance when running BFS on the Twitter and UK graphs. Interestingly, the performance peaks for both graphs when the maximum gap is set to 16 blocks (i.e., 8KB). Graphene also imposes an upper bound for IO size, so that the benefit of IO merging would not be dwarfed by handling of large IO requests. We will discuss this upper bound shortly.

In conclusion, Bitmap provides a very efficient method to manage IO requests for graph processing. We will show later that while the OS already provides similar functionality, this approach is more beneficial for dealing with random IOs to a large amount of data.

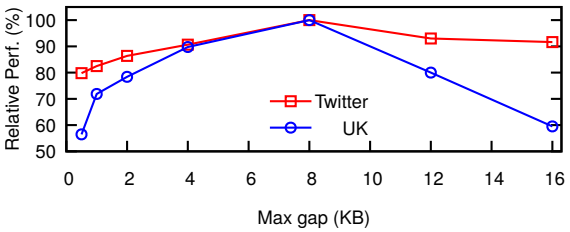
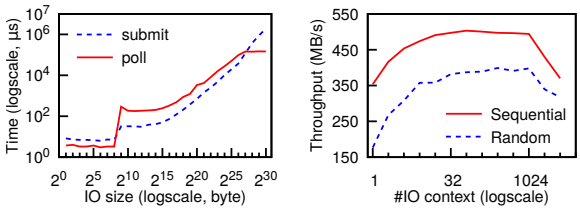


Figure 6: Graphene BFS Performance of maximum gap.



(a) IO size (b) IO context

Figure 7: AIO performance w.r.t. IO size and IO context

Besides Bitmap-based IO, we have also implemented a Pluglist based approach that extends the pluglist to support sorting, deduplication and merging in a global scale. As shown in Section 7, compared to a list, the Bitmap approach incurs smaller overhead and runs four times faster. It is important to note that although we focus on using Bitmap for graph processing in this work, it can also be applied to other applications. We will demonstrate this potential in Section 7.

3.3 Asynchronous IO

Asynchronous IO (AIO) is often used to enable a user-mode thread to read or write a file, while simultaneously carrying out the computation [8]. The initial design goal is to overlap the computation with non-blocking IO calls. However, because graph processing is IO bound, Graphene exploits AIO for a different goal of submitting as many IO requests as possible to saturate the IO bandwidth of flash devices.

There are two popular AIO implementations, i.e., user-level *POSIX AIO* and kernel-level *Linux AIO*. We prefer the latter in this work, because POSIX AIO forks child threads to submit and wait for the IO completion, which in turn has scalability issues while submitting too many IO requests [8]. In addition, Graphene leverages direct IO to avoid the OS-level page cache during AIO, and the possible blocks introduced by the kernel [19].

Upper bound for IO request. Although disks favor large IO sizes in tens or hundreds of MBs, it is not always advantageous to do so, especially for AIO. Typically, an AIO consists of two steps, submitting the IO request to an IO context and polling the context for completion. If IO request sizes are too big, the time for IO submission would take longer than polling, at which point AIO would essentially become blocking IO. Figure 7(a) studies the AIO submission and polling time. As the size goes beyond 1MB, submission time increases quickly. And once it reaches 128MB, it becomes blocked IO as submission time eventually becomes longer then polling time. In this work, we find that a modest IO size, such as 8, 16, and 32 KB, is able to deliver good performance for various graph algorithms. Therefore, we set the default upper bound of IO merging as 16KB.

IO context. In AIO, each IO context loads the IO requests sequentially. Graphene uses multiple contexts to handle the concurrent requests and overlap the IO with the computation. For example, while a thread is working on the request returned from one IO context, another IO context can be used to serve other requests from the same SSD. Given its intensive IO demand, graph computation would normally need to create a large number of IO contexts. However, without any constraints, too many IO contexts would hurt the performance because every context needs to register in the kernel and may lead to excessive overhead from polling and management.

Figure 7(b) evaluates the disk throughput with respect to the number of total IO contexts. As one can see that each SSD could achieve the peak performance with 16 contexts but the performance drops once the total IO context goes beyond 1,024 contexts. In this work, depending on the number of available SSDs, we utilize different numbers of IO contexts, by default using 512 contexts for 16 SSDs.

3.4 Conclusion

In summary, combining 512-byte block and Bitmap-based IO management allows Graphene to load a smaller amount of data from SSDs, about 21% less than the traditional approach. Together with AIO, Graphene is able to achieve high IO throughput of upto 5GB/s for different algorithms on an array of SSDs.

4 Balancing Data and Workload

Taking care of graph data IO only solves half of the problem. In this section, we present data partitioning and workload balancing in Graphene.

4.1 Row-Column Balanced 2D Partition

Given highly skewed degree distribution in power-law graphs, existing graph systems, such as GridGraph [63], TurboGraph [25], FlashGraph [62], and PowerGraph [20], typically apply a simple 2D partitioning method [9] to split the neighbors of each vertex across multiple partitions. The method is presented in Figure 8(a), where each partition accounts for an equal range of vertices, P number of vertices in this case, on both row and column-wise. This approach needs to scan the graph data once to generate the partitions. The main

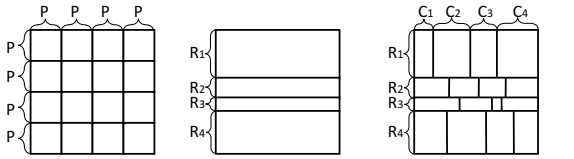


Figure 8: Graphene Balanced 2D partition.

drawback of this approach is that an equal range of vertices in each data partition do not necessarily lead to an equal amount of edges, which can result in workload imbalance for many systems.

To this end, Graphene introduces a row-column balanced 2D partitioning method, as shown in Figure 8(b-c), which ensures each partition contains an equal number of edges. In this case, each partition may have different numbers of rows and columns. This is achieved through three steps: (1) the graph is divided by the row major into R number of partitions, each of which has the same numbers of edges with potentially different number of rows; (2) Each row-wise partition is further divided by the column major into C number of (smaller) partitions, each of which again has the equal amount of edges. As a result, each partition may contain different number of rows and columns. Although it needs to read the graph one more time, it produces "perfect" partitions with the equal amount of graph data, which can be easily distributed to a number of SSDs.

Figure 9 presents the benefits of row-column balanced 2D partition for two social graphs, Twitter and Friendster. On average, the improvements are 2.7× and 50% on Twitter and Friendster, respectively. The maximum and minimum benefits for Twitter are achieved on SpMV for 5× and k-Core 12%. The speedups are similar for Friendster. While each SSD holds a balanced data partition, the workload from graph processing is not guaranteed to be balanced. Rather, the computation performed on each partition can vary drastically depending on the specific algorithm. In the following, we present the workflow of Graphene and how it balances the IO and processing.

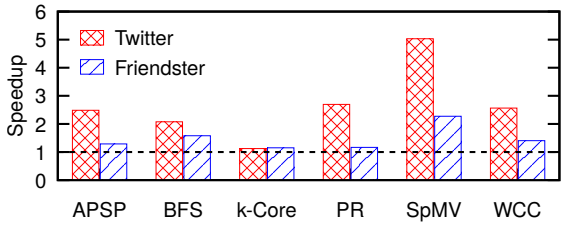


Figure 9: Benefit of row-column balanced 2D partition.

4.2 Balancing IO and Processing

Although AIO, to some extent, enables the overlapping between IO and computation, we have observed that a single thread doing both tasks would fail to fully saturate

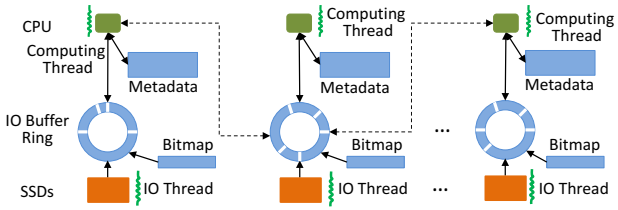


Figure 10: Graphene scheduling management.

the bandwidth of an SSD. To address this problem, one can assign multiple threads to work on a single SSD in parallel. However, if each thread would need to juggle IO and processing, this can lead to contention in the block layer, resulting in a lower performance.

In Graphene, we assign two threads to collaboratively handle the IO and computation on each SSD. Figure 10 presents an overview of the workflow. Initially upon receiving updates to the Bitmap, a dedicated *IO thread* formulates and submits IO requests to the SSD. Once the data is loaded in memory, the *computing thread* retrieves the data from the *IO buffer* and works on the corresponding metadata. Using PageRank as an example, for currently active vertices, the IO thread would load their in-neighbors (i.e., the vertices with a directed edge to active vertices) in the IO buffer, further store them in the ring buffer. Subsequently, the computing thread uses the rank values of those in-neighbors to update the ranks of active vertices. The metadata of interest here is the rank array.

Graphene pins IO and computing threads to the CPU socket that is close to the SSD they are working on. This NUMA-aware arrangement reduces the communication overhead between IO thread and SSD, as well as IO and computing threads. Our test shows that this can improve the performance by 5% for various graphs.

Graphene utilizes a work stealing technique to mitigate computational imbalance issue. As shown in Figure 10, each computing thread first works on the data in its own IO buffer ring. Once it finishes processing its own data, this thread will check the IO buffer of other computing threads. As long as other computing threads have unprocessed data in IO buffers, this thread is allowed to help process them. This procedure repeats until all data have been consumed.

Figure 11 presents the performance benefit from work stealing. On average, PageRank, SpMV, WCC and APSP

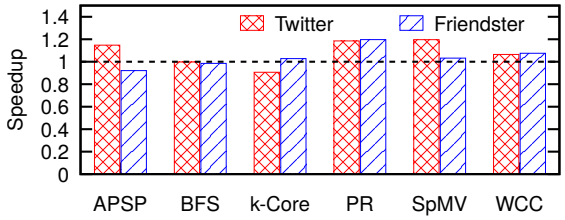


Figure 11: Benefit of workload stealing.

achieve various speedup of 20%, 11%, 8% and 4%, respectively, compared to the baseline of not using workload stealing. On the other hand, BFS and k-Core suffer slowdown of 1% and 3%. This is mostly because the first four applications are more computation intensive while BFS and k-Core are not. One drawback of workload stealing is lock contention at the IO buffer ring, which can potentially lead to performance degradation, e.g., 8% for APSP on Friendster and k-Core on Twitter.

5 HugePage Support

Graphene leverages the support of *Direct HugePages* (DHP), which preallocates hugepages at boot time, to store and manage graph data and metadata structures, e.g., IO buffer and Bitmap, shown as blue boxes in Figure 10. This is motivated by our observation of high TLB misses, as the number of memory pages continues to grow for large-scale graph processing. Because a TLB miss typically requires hundreds of CPU cycles for the OS to go through the page table to figure out the physical address of the page, this would greatly lower the graph algorithm performance.

In Graphene, the OS creates and maintains a pool of hugepages at machine boot time when memory fragmentation is at the minimum. This is because any memory fragmentation would break physical space into pieces and disrupt the allocation of hugepages. We choose this approach over transparent hugepage (THP) in Linux [39] for a couple of reasons. First, we find that THP introduces undesirable uncertainty at runtime, because such a hugepage could be swapped out from memory [42]. Second, THP does not always guarantee successful allocation and may incur high CPU overhead. For example, when there were a shortage, the OS would need to aggressively compress the memory in order to provide more hugepages [54].

Data IO. Clearly, if each IO request were to consume one hugepage, a large portion of memory space would be wasted, because Graphene, even with IO merging, rarely issues large (2MB) IO requests. Alternatively, Graphene allows multiple IO requests to share hugepages. This consolidation is done through IO buffers in the IO Ring Buffer. Given a batch of IO requests, Graphene first claims a buffer that contains a varied number of continuous 2MB hugepages. As the IO thread works exclusively with a buffer, all IO requests can in turn use any portion of it to store the data. Also, consecutive IO requests will use continuous memory space in the IO buffer so that there is no fragmentation. Note that the system needs to record the begin position and length of each request within the memory buffer, which is later parsed and shared with the user-defined Compute function in the IoIterator. In addition, *direct IO* is utilized for loading disk blocks directly into hugepages. Comparing to buffered IO, this method skips the step of copying data to system pagecache and further to user buffer, i.e., double copy.

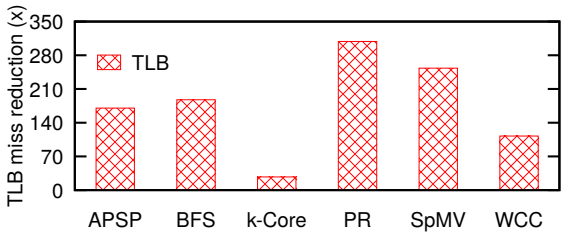


Figure 12: TLB misses reduced by hugepage-enabled buffer.

Metadata has been the focus of several prior works [9, 12, 59] to improve the cache performance of various graph algorithms. As a first attempt, we have investigated the use of page coloring [16, 60] to resolve cache contention, that is, to avoid multiple vertices being mapped to the same cache line. With 4KB pages, we are able to achieve around 5% improvement across various graphs. However, this approach becomes incompatible when we use 2MB hugepages for metadata, as the number of colors is determined by the LLC size (15MB), associativity (20) and page size.

To address this challenge, we decide to use hugepages for the metadata whose size is at the order of $O(|V|)$. In this work, we use 1GB hugepages, e.g., for PageRank, a graph with one billion vertices will need 4GB memory for metadata, that is, four 1GB hugepages.

This approach brings several benefits. Figure 12 illustrates the reduction in TLB miss introduced by this technique when running on a Kronecker graph. Across six algorithms, we observe an average 177× improvement with the maximum of 309× for PageRank. In addition, as prefetching is constrained by the page size, hugepages also enables more aggressive hardware prefetching in LLC, now that the pages are orders of magnitude bigger (1GB vs. 4KB). The test shows that this technique provides around 10% speedup for these graph algorithms.

6 Graph Algorithms

Graphene implements a variety of graph algorithms to understand different graph data and metadata, and their IO patterns. For all the algorithms, the sizes of data and metadata are $O(|E|)$ (total count of edges) and $O(|V|)$ (total count of vertices), respectively.

Breadth First Search (BFS) [4, 33] performs random reads of the graph data, determined by the set of most recently visited vertices in the preceding level. The statuses (visited or unvisited) of the vertices are maintained in the status array, a key metadata in BFS. It is worthy to note that status array may experience more random IOs, because the neighbors for a vertex tend to have different IDs, some of which are far apart.

PageRank (PR) [26,41] can calculate the popularity of a vertex by either pulling the updates from its in neighbors

or pushing its rank to out neighbors. The former performs random IO on the rank array (metadata), whereas the latter requires sequential IO for graph data but needs locks while updating the metadata. In this work, we adapt delta-step PageRank [61], where only vertices with updated ranks should push their delta values to the neighbors, yet again requiring random IOs.

Weakly Connected Component (WCC) is a special type of subgraph whose vertices are connected to each other. For directed graphs, a strongly connected component exists if a directed path can be found between all pairs of vertices in the subgraph [28]. In contrast, a WCC exists if such a path can be found regardless of the edge direction. We implement the hybrid WCC detection algorithm presented in [48], that is, it uses BFS to detect the largest WCC then uses label propagation to compute remaining smaller WCCs. In this algorithm, the label array serves as the metadata.

k-Core (KC) [37, 45] is another type of subgraph where each vertex has the degree of at least k. Iteratively, a k-Core subgraph is found by removing the vertices from the graph whose degree is less than k. As the vertices are removed, their neighbors are affected, where the metadata – degree array – will need to be updated. Similar to aforementioned algorithms, since the degree array is indexed by the vertex IDs, the metadata IO in k-Core also tends to be random. k-Core is chosen in this work as it presents alternating graph data IO patterns across different iterations. Specifically, in the initial iterations, lots of vertices would be affected when a vertex is removed, thus the graph data is retrieved likely in the sequential order. However at the later iterations, fewer vertices will be affected, resulting in random graph data access.

All Pairs Shortest Path (APSP) calculates the shortest paths from all the vertices in the graph. With APSP, one can further compute Closeness Centrality and Reachability problems. Graphene combines multi-source traversals together, to reduce the total number of IOs needed during processing and the randomness exposed during the metadata access [34, 51]. Similar to FlashGraph, we randomly select 32 source vertices for evaluation to reduce APSP execution time on large graphs.

Sparse Matrix Vector (SpMV) multiplication exhibits sequential access when loading the matrix data, and random access for the vector. In this algorithm, the matrix and vector serve the role as graph data and metadata, respectively. As a comparison to BFS, SpMV is more IO friendly but equally challenging on cache efficiency.

7 Evaluations

We have implemented a prototype of Graphene in 3,300 lines of C++ code, where the IoIterator accounts for 1,300 lines and IO functions 800 lines. Six graph algorithms are implemented with average 200 lines of code. We perform our experiments on a server with a dual-socket Intel Xeon E5-2620 processor (total 12 cores and 24 threads with hyperthreading), 128GB memory, 16 500GB Samsung 850 SSDs connected with two LSI SAS 9300-8i host bus adapters, and Linux kernel 4.4.0.

Table 2: Graph Datasets.

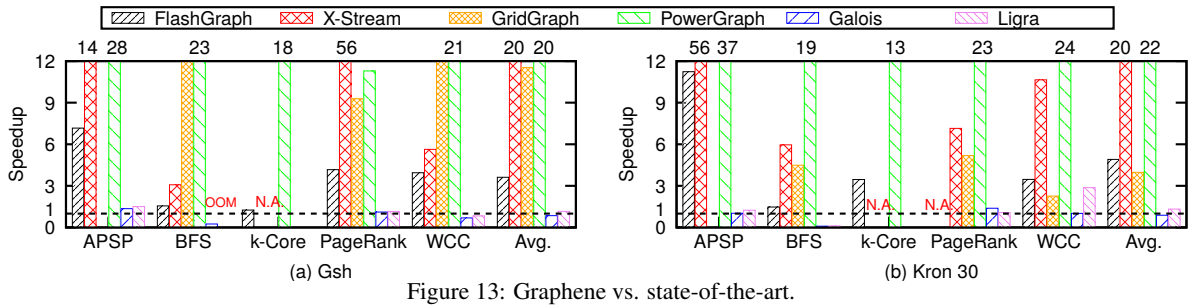
Name	# Vertices	# Edges	Size	Preprocess (seconds)
Clueweb	978M	42.6B	336GB	334
EU	1071M	92B	683GB	691
Friendster	68M	2.6B	20GB	3
Gsh	988M	33.8B	252GB	146
Twitter	53M	2.0B	15GB	2
UK	788M	48B	270GB	240
Kron30	1B	32B	256GB	141
Kron31	2B	1T	8TB	916

Table 2 lists all the graphs used in this paper. Specifically, Twitter [2] and Friendster [1] are real-world social graphs. In particular, Twitter contains 52,579,682 vertices and 1,963,263,821 edges, and Friendster is an online gaming network with 68,349,466 vertices and 2,586,147,869 edges. In addition, Clueweb [13], EU [18], Gsh [23] and UK [55] are webpage based graphs provided by webgraph [5–7]. Among them, EU is the largest with over one billion of vertices and 90 billion of edges. On the other hand, two Kronecker graphs are generated with the Graph500 generator [22] with scale 30 and 31, which represent the number of vertices as 1 billion (2^{30}) and 2 billion (2^{31}), with number of edges of 32 billion and 1 trillion. This paper, by default uses 8 bytes to represent a vertex ID unless explicitly noted. We run the tests five times and report the average values.

In addition, Table 2 presents the time consumption of the preprocessing step of the row-column balanced 2D partition. On average, our partition method takes 50% longer time than the conventional 2D partition method, e.g., preprocessing the largest Kron31 graph takes 916 seconds. Note that except X-Stream, many graph systems, including FlashGraph, GridGraph, PowerGraph, Galois and Ligra, also require similar or longer preprocessing to prepare the datasets. In the following, we report the runtime of graph algorithms, excluding the preprocessing time for all graph systems.

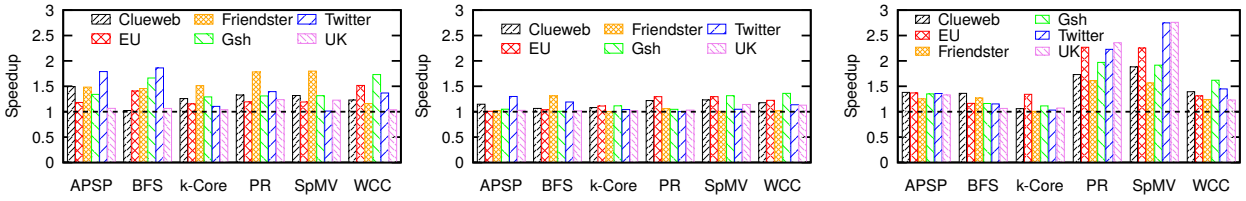
7.1 Comparison with the State of the Art

We compare Graphene against FlashGraph (semi-external memory), X-Stream (external memory), GridGraph (external memory), PowerGraph (in-memory), Galois (in-memory), and Ligra (in-memory) when running various algorithms. Figure 13 reports the speedup of Graphene over different systems for all five algorithms. SpMV is currently not supported in other systems except our Graphene, and k-Core is only provided by FlashGraph, PowerGraph and Graphene. In the figure the label "NA" indicates lack of support in the system. In this test, we choose one real graph (Gsh) and one synthetic graph



(a) Gsh (b) Kron 30

Figure 13: Graphene vs. state-of-the-art.



(a) Bitmap vs. Pluglist (b) HugePage vs. 4K Page (c) Dedicated IO and computing threads

Figure 14: Overall performance benefits of IO techniques.

(Kron30). Note that Gsh is the largest graph that is supported by in-memory systems. We have observed similar performance on other graphs.

In general, Graphene outperforms external memory systems FlashGraph, GridGraph and X-Stream by 4.3×, 7.8× and 20×, respectively. Compared to in-memory systems PowerGraph, Galois and Ligra where all graph data are stored in DRAM, Graphene keeps the data on SSDs and reads on-demand, outperforming PowerGraph by 21× and achieving a comparable performance with the other two (90% for Galois and 1.1× for Ligra). Excluding BFS which is the most IO intensive and favors in-memory data, Graphene outperforms Galois and Ligra by 10% and 45%, respectively. We also compare Graphene with an emerging Differential Dataflow system [53] and Graphene is able to deliver an order of magnitude speedup on BFS, PageRank and WCC.

For the Gsh graph, as shown in Figure 13, Graphene achieves better performance than other graph systems for different algorithms with exceptions for BFS and WCC. For example, for APSP, Graphene outperforms PowerGraph by 29×, Galois by 35%, Ligra by 50%, FlashGraph by 7.2× and X-Stream by 14×. For BFS and WCC, Graphene runs faster than GridGraph, PowerGraph, FlashGraph and X-Stream, but is slower than the two in-memory systems, mostly due to relatively long access latency on SSDs compared to DRAM. Similar performance benefits can also be observed on the syntheic Kron30 graph.

Table 3: Graphene runtime on Kron31 (seconds).

Name	APSP	BFS	k-Core	PageRank	WCC	SpMV
Kron31	7,233	2,630	318	25,023	3,023	5,706

Trillion-edge graph. We further evaluate the performance of Graphene on Kron31 as presented in Table 3. On average, all algorithms take around one hour to finish, with the maximum from PageRank of 6.9 hours while k-Core can be completed in 5.3 minutes. To the best of our knowledge, this is among the first attempts to evaluate trillion-edge graphs on a external-memory graph processing system.

7.2 Benefits of IO Techniques

This section examines the impacts on the overall system performance brought by different techniques independently, including Bitmap, hugepage, and dedicated IO and computing threads. We run all six algorithms on all six real-world graphs.

The Bitmap provides an average 27% improvement over using the pluglist as presented in Figure 14(a). Clearly, Bitmap favors the algorithms with massive random IOs such as WCC and BFS and low diameter graphs such as Gsh, EU, and Friendster. For example, Bitmap achieves about 70% speedup on Gsh on both BFS and WCC, and 30% for other algorithms.

Figure 14(b) compares the performance of hugepages and 4KB pages. Hugepages provides average 12% improvement and the speedup varies from 17% for WCC to 6% for k-Core. Again, two largest improvements are achieved on the (largest) Gsh graph for SpMV and WCC.

The benefit introduced by dedicated IO and computing threads is presented in Figure 14(c), where the baseline is using one thread for both IO and computing. In this case, Graphene achieves an average speedup of 54%. Particularly, PageRank and SpMV enjoy significant higher improvement (about 2×) than the other algorithms.

7.3 Analysis of Bitmap-based IO

We study how Bitmap-based IO affects the IO and computing ratio of different algorithms in Figure 15. Without

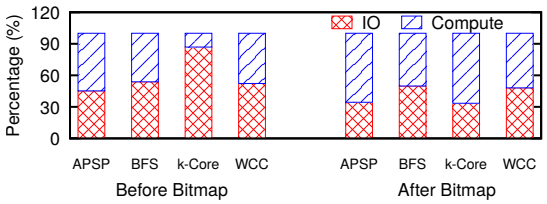
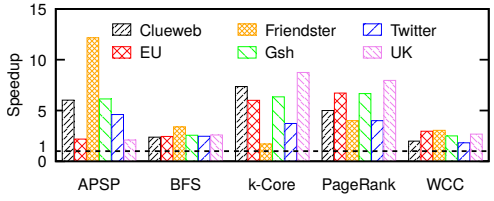


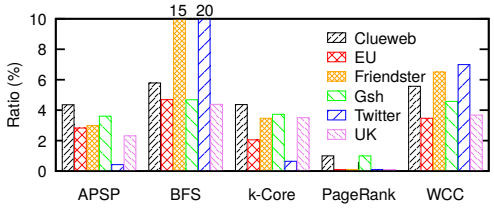
Figure 15: Runtime breakdown of IO and computing with Bitmap-based IO.

bitmap, all four algorithms spend about 60% on IO and 40% on computation. In comparison, the distribution of runtime reverses with bitmap, where computation takes average 60% of the time and IO 40%. Because the IO time is significantly reduced, faster IO as a result accelerates the execution of the algorithms. In particular, the biggest change comes from k-Core where IO accounts for 87% and 34% before and after bitmap.

As shown in Figure 16, when compared to a pluglist-based approach, the Bitmap-based IO runs 5.5×, 2.6×, 5.6×, 5.7× and 2.5× faster on APSP, BFS, k-Core, PageRank, and WCC, respectively. Note that here we only evaluate the time consumption of preparing the bitmap and pluglist, which is different from overall system performance presented in Figure 14. On the other hand, in most cases, adding Bitmap incurs a small increase of about 3.4% of total IO time. However, for a few cases with relatively high overhead, it is most likely caused by the small size of the graph data (e.g., Friendster and Twitter), as well as random IOs of the algorithms (e.g., BFS). The time spent on Bitmap varies from about 60 milliseconds for PR and SpMV (less than 1% of total IO time), to 100 seconds for APSP (2.3% of IO time).



(a) Preparing Bitmap vs. Pluglist.



(b) Overhead.

Figure 16: Bitmap performance and overhead.

Bitmap-based IO can be applied to other applications beyond graph processing. Figure 17 examines the time consumption differences between Bitmap based IO and Linux IO. Here we replay the reads in five IO traces

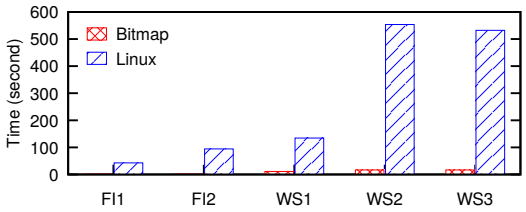
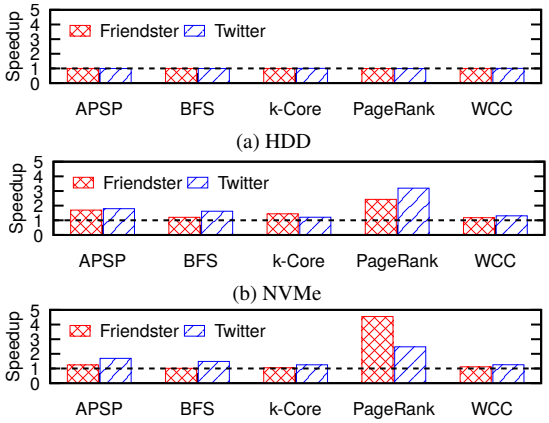


Figure 17: Bitmap-based IO performance on traces.

as quickly as possible, namely Financial 1-2 and Web-Search 1-3 from UMass Trace Repository [3]. On average, the Bitmap is 38× faster than Linux IO, with the maximum speedup of 74× obtained on Financial2 (from 94.2 to 1.26 seconds). The improvement comes mostly from more (9.3×) deduplicated IOs and more aggressive IO merging.

Figure 18 further studies the impacts of bitmap based IO on hard disk (HDD), NVMe and Ramdisk. In this test, we use five Seagate 7200RPM SATA III hard drives in a Raid-0 configuration, and one Samsung 950 Pro NVMe device. One can see that compared to the pluglist based method, although bitmap improves hard disk performance only marginally (1% on average), faster storage devices such as NVMe and Ramdisk are able to achieve about 70% improvement in IO performance.



(a) HDD

(b) NVMe

(c) Ramdisk

Figure 18: Bitmap performance on HDD, NVMe and Ramdisk.

7.4 Scalability, Utility, and Throughput

This section studies the scalability of Graphene with respect to the number of SSDs. Recall that Graphene uses two threads per SSD, one IO and another compute. Using a single thread would fail to fully utilize the bandwidth of an SSD. As shown in Figure 19, Graphene achieves an average 3.3× speedup on the Kron30 graph when scaling from a single SSD (two threads) to eight SSDs (16 threads). Across different applications, SpMV enjoys the biggest 3.7× speedup and PageRank the smallest 2.6×. The small performance gain from 8 to 16 SSDs is due to the shift of the bottleneck from IO to CPU.

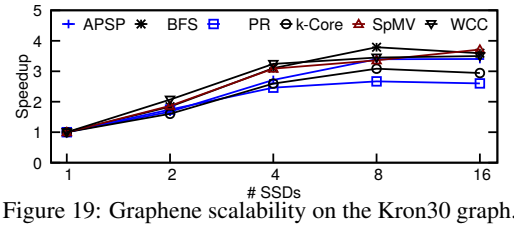


Figure 19: Graphene scalability on the Kron30 graph.

Recall that IO utility is defined as the ratio of useful data and total data loaded, we evaluate the IO utility when using 512-byte IO vs. 4KB IO on various algorithms and graph datasets. As presented in Figure 20, Graphene achieves 20% improvement on average. For APSP and BFS, one can see about 30% improvement with the best benefit of 50% on UK. Similar speedups can also be observed for K-Core and WCC. In contrast, PageRank and SpMV present minimal benefit because the majority of their iterations load the whole graph.

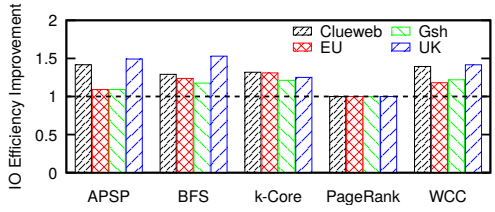


Figure 20: Utility of 512-byte vs. 4KB IO.

To demonstrate the IO loads of different disks in Graphene, we further examine the throughput of 16 SSDs for two applications, BFS and PageRank. Figure 21 show the throughput for the fastest (max) and slowest (min) SSDs, as well as the median throughput. Clearly, the 16 SSDs are able to deliver similar IO performance for most of run, with an average difference of 6 to 15 MB/s (5-7% for PageRank and BFS). For both algorithms, the slowest disk does require extra time to complete the processing, which we leave for future research to close the gap.

8 Related Work

Recent years have seen incredible advances in graph computation, to name a few, in-memory systems [27, 40, 47], distributed systems [10, 11, 20, 38, 46, 61], external-memory processing [21, 25, 31, 32, 35, 36, 43, 44, 57, 62, 63], and accelerator-based systems [30, 33, 58]. In this section, we compare Graphene with existing projects from three aspects: programming, IO, and partitioning.

Programming. Prior projects, regardless of *Think like a vertex* [10, 32, 36, 58], *Think like an edge* [31, 43, 44], *Think like an embedding* [50], or *Think like a graph* [52], center around simplifying computation related programming efforts. In comparison, Graphene aims for ease of IO management with the new IO iterator API.

IO optimization is the main challenge for external

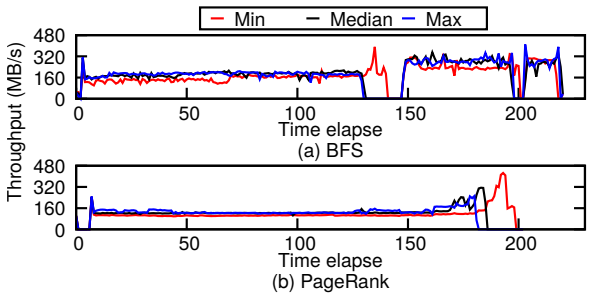


Figure 21: Throughputs of the fastest (max) and slowest (min) SSDs, and median throughput out of 16 SSDs.

memory graph engines, for which Graphene develops a set of fine-grained IO management techniques, including using 512-byte IO block and bitmap-based selective IO. Our approach achieves high efficiency compared to full IO [32, 36, 43, 44]. Compared to GridGraph [63] and FlashGraph [62], Graphene introduces a finer grained method that supports global range IO adjustment and reduces IO requests by 3×. Also, Graphene shows that asynchronous IOs, when carefully managed, are very beneficial for external memory systems. While hugepages are not new to graph systems [40, 62], Graphene addresses the issue of potentially low memory utilization by constructing IO buffers to share hugepages.

Partition optimization. A variety of existing projects [12, 20, 62, 63] rely on conventional 2D partitioning [9] to balance the workload. In contrast, Graphene advocates that it is the amount of edges, rather than vertices, in a partition that determines the workload. The new row-column balanced partition can help achieve up to 2.7× speedup on a number of graph algorithms.

9 Conclusion and Future work

In this paper, we have designed and developed Graphene that consists of a number of novel techniques including IO centric processing, Bitmap-based asynchronous IO, hugepage support, data and workload balancing. It allows the users to treat the data as in-memory, while delivering high-performance on SSDs. The experiments show that Graphene is able to perform comparably against in-memory processing systems on large-scale graphs, and also runs several times faster than existing external-memory processing systems.

10 Acknowledgments

The authors thank the anonymous reviewers and our shepherd Brad Morrey for their valuable suggestions that help improve the quality of this paper. The authors also thank Da Zheng, Frank Mcsherry, Xiaowei Zhu, and Wenguang Chen for their help and discussion. This work was supported in part by National Science Foundation CAREER award 1350766 and grant 1618706.

References

[1] Friendster Network Dataset – KONECT. http://konect.uni-koblenz.de/networks/friendster, 2016.

[2] Twitter (MPI) Network Dataset – KONECT. http://konect.uni-koblenz.de/networks/twitter_mpi, 2016.

[3] UMASS Trace Repository. http://traces.cs.umass.edu/, 2016.

[4] Scott Beamer, Krste Asanović, and David Patterson. Direction-Optimizing Breadth-First Search. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[5] Paolo Boldi, Andrea Marino, Massimo Santini, and Sebastiano Vigna. BUbiNG: Massive Crawling for the Masses. In *Proceedings of the Companion Publication of the 23rd International Conference on World Wide Web (WWW)*, 2014.

[6] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks. In *Proceedings of the 20th international conference on World Wide Web (WWW)*, 2011.

[7] Paolo Boldi and Sebastiano Vigna. The WebGraph Framework I: Compression Techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW)*, Manhattan, USA, 2004.

[8] Daniel Bovet and Marco Cesati. *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005.

[9] Aydin Buluç and Kamesh Madduri. Parallel Breadth-First Search on Distributed Memory Systems. In *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.

[10] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. Powerlyra: Differentiated Graph Computation and Partitioning on Skewed Graphs. In *Proceedings of the Tenth European Conference on Computer Systems (Eurosys)*, 2015.

[11] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. Kineograph: Faking the Pulse of A Fast-Changing and Connected World. In *Proceedings of the european conference on Computer Systems (Eurosys)*, 2012.

[12] Jatin Chhugani, Nadathur Satish, Changkyu Kim, Jason Sewall, and Pradeep Dubey. Fast and Efficient Graph Traversal Algorithm for CPUs: Maximizing Single-Node Efficiency. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2012.

[13] Clueweb dataset from WebGraph. http://law.di.unimi.it/webdata/clueweb12/, 2012.

[14] Thayne Coffman, Seth Greenblatt, and Sherry Marcus. Graph-Based Technologies For Intelligence Analysis. *Communications of the ACM*, 2004.

[15] Antonio Del Sol, Hirotomo Fujihashi, and Paul O'Meara. Topology of Small-World Networks of Protein-Protein Complex Structures. *Bioinformatics*, 2005.

[16] Xiaoning Ding, Kaibo Wang, and Xiaodong Zhang. ULCC: A User-Level Facility for Optimizing Shared Cache Performance on Multicores. In *Proceedings of the SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*, 2011.

[17] Christian Doerr and Norbert Blenn. Metric Convergence in Social Network Sampling. In *Proceedings of the 5th ACM workshop on HotPlanet*, 2013.

[18] EU dataset from WebGraph. http://law.di.unimi.it/webdata/eu-2015/, 2015.

[19] Fixing asynchronous I/O, again. https://lwn.net/Articles/671649/, 2016.

[20] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2012*.

[21] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. GraphX: Graph Processing in a Distributed Dataflow Framework. In *Proceedings of the USENIX conference on Operating Systems Design and Implementation (OSDI)*, 2014.

[22] Graph500. http://www.graph500.org/.

[23] Gsh dataset from WebGraph. http://law.di.unimi.it/webdata/gsh-2015/, 2015.

[24] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran,

Wenguang Chen, and Enhong Chen. Chronos: A Graph Engine For Temporal Graph Analysis. In *Proceedings of the european conference on Computer systems (Eurosys)*, 2014.

[25] 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 SIGKDD international conference on Knowledge discovery and data mining (KDD)*, 2013.

[26] Taher H Haveliwala. Topic-Sensitive Pagerank. In *Proceedings of the 11th international conference on World Wide Web (WWW)*, 2002.

[27] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. Green-Marl: A DSL For Easy and Efficient Graph Analysis. In *Proceedings of the international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[28] Sungpack Hong, Nicole C Rodia, and Kunle Olukotun. On Fast Parallel Detection of Strongly Connected Components (SCC) in Small-World Graphs. In *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2013.

[29] Hawoong Jeong, Sean P Mason, A-L Barabási, and Zoltan N Oltvai. Lethality and Centrality in Protein Networks. *Nature*, 2001.

[30] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N Bhuyan. CuSha: Vertex-Centric Graph Processing on GPUs. In *Proceedings of the international symposium on High performance distributed computing (HPDC)*, 2014.

[31] Pradeep Kumar and H Howie Huang. 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)*, 2016.

[32] Aapo Kyrola, Guy E Blelloch, and Carlos Guestrin. GraphChi: Large-Scale Graph Computation on Just a PC. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2012*.

[33] Hang Liu and H Howie Huang. Enterprise: Breadth-First Graph Traversal on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2015.

[34] Hang Liu, H. Howie Huang, and Yang Hu. iBFS: Concurrent Breadth-First Search on GPUs. In *Proceedings of the SIGMOD International Conference on Management of Data (SIGMOD)*, 2016.

[35] 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*, 2012.

[36] 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 SIGMOD International Conference on Management of data (SIGMOD)*, 2010.

[37] Alberto Montresor, Francesco De Pellegrini, and Daniele Miorandi. Distributed k-Core Decomposition. *IEEE Transactions on Parallel and Distributed Systems*, 2013.

[38] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A Timely Dataflow System. In *Proceedings of the Twenty-Fourth Symposium on Operating Systems Principles (SOSP)*, 2013.

[39] Juan Navarro, Sitararn Iyer, Peter Druschel, and Alan Cox. Practical, Transparent Operating System Support for Superpages. In *Proceedings of the 5th symposium on Operating systems design and implementation (OSDI)*, 2002.

[40] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A Lightweight Infrastructure for Graph Analytics. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, 2013.

[41] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank Citation Ranking: Bringing Order To the Web. In *Stanford InfoLab Technical Report*, 1999.

[42] Performance Issues with Transparent Huge Pages (THP). `https://blogs.oracle.com/linux/entry/performance_issues_with_transparent_huge`, 2013.

[43] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. Chaos: Scale-out Graph Processing from Secondary Storage. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*, 2015.

[44] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-Stream: Edge-centric Graph Processing using Streaming Partitions. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP), 2013*.

[45] Ahmet Erdem Sariyüce, Buğra Gedik, Gabriela Jacques-Silva, Kun-Lung Wu, and Ümit V Çatalyürek. Streaming Algorithms for k-Core Decomposition. *Proceedings of the VLDB Endowment*, 2013.

[46] Bin Shao, Haixun Wang, and Yatao Li. Trinity: A Distributed Graph Engine on a Memory Cloud. In *Proceedings of the SIGMOD International Conference on Management of Data (SIGMOD)*, 2013.

[47] Julian Shun and Guy E Blelloch. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*, 2013.

[48] George M Slota, Sivasankaran Rajamanickam, and Kamesh Madduri. BFS and Coloring-Based Parallel Algorithms For Strongly Connected Components and Related Problems. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2014.

[49] Samsung 850 EVO SSD. `http://www.samsung.com/semiconductor/minisite/ssd/product/consumer/850evo.html`, 2015.

[50] 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 (SOSP)*, 2015.

[51] Manuel Then, Moritz Kaufmann, Fernando Chirigati, Tuan-Anh Hoang-Vu, Kien Pham, Alfons Kemper, Thomas Neumann, and Huy T Vo. The More the Merrier: Efficient Multi-Source Graph Traversal. *Proceedings of the VLDB Endowment*, 2014.

[52] 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*, 2013.

[53] Timely Dataflow Blog. `https://github.com/frankmcsherry/timely-dataflow`, 2016.

[54] Transparent huge pages in 2.6.38. `http://lwn.net/Articles/423584/`, 2011.

[55] UK dataset in WebGraph. `http://law.di.unimi.it/webdata/uk-2014/`, 2014.

[56] Keval Vora, Guoqing Xu, and Rajiv Gupta. Load the Edges You Need: A Generic I/O Optimization for Disk-based Graph Processing. In *USENIX Annual Technical Conference (ATC)*, 2016.

[57] Kai Wang, Guoqing Xu, Zhendong Su, and Yu David Liu. GraphQ: Graph Query Processing with Abstraction Refinement—Scalable and Programmable Analytics over Very Large Graphs on a Single PC. In *Proceedings of the Usenix Annual Technical Conference (ATC)*, 2015.

[58] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. Gunrock: A High-Performance Graph Processing Library on the GPU. In *Proceedings of SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2016.

[59] Ming Wu, Fan Yang, Jilong Xue, Wencong Xiao, Youshan Miao, Lan Wei, Haoxiang Lin, Yafei Dai, and Lidong Zhou. GRAM: Scaling Graph Computation to the Trillions. In *Proceedings of the Sixth Symposium on Cloud Computing (SoCC)*, 2015.

[60] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. Towards Practical Page Coloring-Based Multicore Cache Management. In *Proceedings of the European conference on Computer systems (Eurosys)*, 2009.

[61] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. Maiter: An Asynchronous Graph Processing Framework For Delta-Based Accumulative Iterative Computation. *IEEE Transactions on Parallel and Distributed Systems*, 2014.

[62] 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 *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, 2015.

[63] Xiaowei Zhu, Wentao Han, and Wenguang Chen. GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning. In *USENIX Annual Technical Conference (ATC)*, 2015.