# GraphMap: Investigating Locality Improvements in Graph Applications Through Partitioning

Coulter Beeson
cdbees@cs.ubc.ca

Laura Cang
cang@cs.ubc.ca

Meghana Venkataswamy
meghanav@cs.ubc.ca

December 13, 2017

## Abstract

We investigate the effect of graph partitioning on data locality. To improve the performance of specific graph-based applications, we developed a system as well as a custom graph file type we collectively call GraphMap. We evaluated the space requirements in graph creation as well as the page fault rate. To measure the effect of coupling partitions with applications, a static evaluation was performed that found that structured partitions outperform graph agnostic partitioning. Finally we endeavoured to create a system that could dynamically partition a graph during the execution of the application, however this extension remains in production.

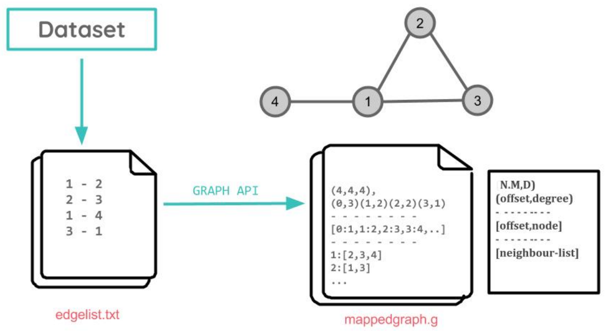


Figure 1: The data structure for GraphMap relies on fixed-length neighbour list to simplify vertex-swapping

## 1 Introduction

As compute resources and storage become cheaper and data collection more pervasive, large scale graph structures are becoming exceedingly common. For instance, the Facebook friend graph has approximately 2 billion active monthly users with an average of 338 friends each [1]for an estimated 338 billion edges, if including pages "liked" this grows to over 1 trillion edges. When performing a graph-wide analysis on these structures, the entire graph may not fit in memory and significant overhead is incurred during paging due to poor locality of the graph structure.

Many graph algorithms traverse the graphs in very predictable ways, often preempting memory access of neighbouring nodes both temporally and proximally [6]. By leveraging access patterns, we hope that mapping nodes to memory will improve spatial locality and therefore, performance of large graph traversal algorithms. Using a partitioning algorithm to localize nodes that are commonly accessed together on a small number of pages can improve hit rates [4, 5].

Determining specific access patterns for an algorithm involves traversing the graph at least in part. However, if the graph is largely static and an application with a similar access pattern is used repeatedly the overhead cost of pre-partitioning the graph may be amortized and performance improvements significant in the long run. We investigate the costs of partitioning a graph over the course of the first traversal iteration.

We propose a custom graph structure that aids locality by allowing for easy swapping of node entries in mapped memory to facilitate partitioning onto pages (see Figure 1). To accomplish this, we designed our graph API to populate a 4K page with an integer number of nodes: each node occupies a fixed-width line, rounded up to a power of 2, to ensure that node size is a factor of page size. Node access becomes simple as only an offset is needed to find a specific adjacency list. This permits swapping the location of two nodes efficiently. To test this, we first implemented a conversion program that generates a custom

[1]Most recent stats referenced `https://techcrunch.com/2017/06/27/facebook-2-billion-users/`

graph object (my__graph.g) from an edge list representation of a graph (provided by the Stanford Large Network Dataset Collection [1] —a collection of web-based graph data that is anonymized and publicly available). We hoped to improve locality by running partition algorithms that re-order data and then testing to determine page-fault behaviour under a number of traversal patterns. We evaluated performance of statically generated partitions by counting page faults incurred on pre-generated node orderings, comparing to optimal orderings (e.g. BFS traversal on data partitioned in BFS order) against more realistic data. We also present the design of dynamic partitioning where a page fault triggers our handler with partitioning performed on-the-fly before servicing the page request. Implementation continues on this feature as our investigation shows promise and offers a great deal of extensibility. However, as our later sections highlight, we discovered that getting this portion right would entail significant effort restructuring of the graph file; thus we have forgone a less meaningful evaluation on partially working code in favour of presenting our design recommendations and lessons learned throughout this endeavour.

Over the course of this project, we have become more comfortable with a number of concepts introduced in class and, without directly aiming to, developed a deeper familiarity with C and its low level memory manipulation features. Our contributions are as follows:

- a custom file type and graph API, including all code (`http://bit.ly/2BfYDrC`)
- static evaluations of space efficiency and partitioning effects
- a summary of pitfalls and triumphs

The subsequent sections will describe our implementation and accompanying evaluation. We reflect on the design inspirations (Related Work), implications, and limitations (Discussion) and conclude with key lessons learned from this project.

## 2 System Architecture

When conceptualizing our project we broke it down into four sections of implementation:

- Graph File/API
- Pre-processing Edge Lists to Graphs
- Static Partitions and Traversals
- Dynamic Partitioning

### 2.1 Graph File/API

The graph objects used throughout this project are essentially just memory mapped files with an accompanying graph API. Fundamentally our approach uses an adjacency list structure coupled with a header section to store additional node values and indices (see Figure 1. Specifically our file starts with three integers representing an upper bound on the number of nodes $N$, the actual number of edges $M$, and an upper bound on the maximum degree of the graph $D$ rounded up to the nearest power of two. By setting upper bounds on the number of nodes and the maximum degree, it allows us to pre-allocate all of the space needed for the rest of the graph file. Following the $N$, $M$, $D$ in the file is a list of nodes where each node has a corresponding index into the list of adjacency lists and its actual degree (in this model we make no firm distinction between a node with no neighbours and a node absent from the graph completely. Support for this could be added by reserving the zero'th index as a place holder for absent nodes). Following the index-degree pairs is a list of nodes indexed by their offsets allowing faults at specific addresses to be translated back into a fault on a specific node. Together these three components represent the header section that is then padded so that the header fits cleanly on a page boundary.

Following the header section are the adjacency lists for each node. Notably by forcing the upper bound on the degree for each node to be a power of two less than or equal to the page size means that each page is filled with an integral number of adjacency lists, and by forcing the length of all the adjacency lists to be the same means that the location of any two nodes can be swapped simply by overwriting their lists and changing their offsets in the header section. Swaps were initially performed lazily where trailing zeros on a line were not copied over, this was changed last minute as it made debugging slightly harder, but could provide significant performance benefits when swapping nodes with low degree. Simple arithmetic can be performed to map offsets to specific pages and thus swapping nodes can be used to partition nodes on to specific pages. This representation, while much more space efficient than an adjacency matrix for sparse graphs, does potentially waste a large amount of space which is investigated further during our evaluation (refer to Figure 1 for a graphical representation).

## 2.2 Pre-processing

Pre-processing was only necessary to put the edge list data acquired from the Stanford Large Network Dataset Collection into a standardized form complete with the bounds of the size of the graph prior to constructing the graph file proper. While this represented an unexpectedly large portion of the implementation effort, it could be avoided if the data were generated in a common format, and by extending the graph interface to accommodate dynamically re-sizing of the graph as edges are added.

## 2.3 Static Analysis and Partitions

The key idea of this project lies in partitioning methods; we describe our approach and root our rationalization in the relevant theory.

### 2.3.1 Theoretical Background

Graph partitioning has a long and rich theoretical background and has been used to improve graph processing performance in many domains, especially in portioning out graphs to machines for distributed processing as well as numerous other applications.

Formally graph partitioning is to separate a graph into some number of disjoint subsets such that a given metric (usually the cut metric) is minimized. When the number of partitions is two, this problem is polytime solvable using network flow approaches as it can easily be seen to be the minimum cut problem. However if one also imposes a balance constraint that all the partitions are (roughly) equal in size this problem becomes NP-Hard. This problem is fixed parameter tractable in theory when parameterizing by the maximum degree although to be optimally solved requires $O(n^{k^2})$ time. In fact if an exact balance constraint is imposed this problem is inapproximable, and is only approximable up to a logarithmic factor when relaxing to $\epsilon$-balanced partitions (unless $P = NP$). Our specific variant is to partition the graph across pages where the page size cannot be exceeded (and using less than a full page is wasteful) hence this is a variant of $k$-way balanced graph partitioning.

Despite the existence of approximation algorithms for partitioning most approaches are still infeasible on large graphs often taking as much time to pre-process as the final application takes to run. Most real world approaches to graph partition use heuristics. The number and depth of approaches is far too great to be detailed in this paper, but the motivated reader is encouraged to read the survey by Buluç et. al [2].

### 2.3.2 Partitions and Traversals

In our system we are interested in minimizing the number of page faults for a specific application rather than the cut metric. One can easily see that if the access order of an application is known ahead of time, such an access order could be used as an oracle to perform optimal partitioning for that application. While requiring the access order for an application prior to running an application creates a circular dependency, many algorithms follow very similar access patterns (at least approximately) such as depth or breadth first traversals. For single pass algorithms the traversal order can directly be used to place nodes on pages and thus can viewed as a partitioning of the graph. It follows that if you can pre-characterize your application as following a known pattern of access, very simple partitions exist.

To test this idea we implemented both a breadth-first and a depth-first traversal that were used to partition graphs against a base line, random, to represent a graph that has been built without partitioning in mind.

## 2.4 Dynamic Partitioning

To dynamically partition our graph and to track paging statistics we opted to use the userfaultfd interface at the suggestion of members of the NSS lab. The model is quite simple where an application is run in a dedicated thread that first opens an anonymous region of memory the same size as the graph file and registers the adjacency list region with userfaultfd. This anonymous region is then treated as a graph object which is passed to the application. After the memory is registered, a handler thread with access to the real graph file is launched and listens for page faults from the application. When a fault is detected, the handler records statistics and then has the opportunity to perform partitioning or just service the page fault from the graph file directly. This partitioning persists for subsequent runs. The framework is both easy to understand as well as extensible. One could imagine partitioning approaches that are learned as the application executes, adjusts based on performance measures (for instance on different regions of the graph), or receiving advice from the application as the traversal progresses. While this is the portion of the project we were most excited to explore it unfortunately remains non-functional due to issues of keeping the headers of the application graph and the real graph consistent. We discuss this file structure design choice and contemplate alternatives in Section 4.

Table 1: Space costs during graph conversion on various datasets

| | # vertices | # edges | max degree | file (size) | graph (size) |
|---|---|---|---|---|---|
| Friendster | 65,608,366 | 1,806,067,135 | n/a | 20 Gb | n/a |
| YouTube | 1,134,890 | 2,987,624 | 28,754 | 38Mb | 200Gb |
| Amazon | 334,863 | 925,872 | 549 | 13Mb | 1.4Gb |
| DBLP | 317,080 | 1,049,866 | 343 | 14Mb | 700Mb |
| Facebook | 4,039 | 88,234 | 1,045 | 921Kb | 65Mb |

## 3 Evaluation

Our evaluation is performed in two parts: space usage and page faults. First, we examine the space costs of our graph by comparing the ratio of the size of the final graph object versus the size of its generating edge list. To demonstrate the variation in the amount of wasted space, we chose to measure the size increase for a complete graph (which is efficient) and a star graph (which is inefficient), we also use the Facebook dataset as a real world example [2]as provided by SNAP [1].

Second, we use the same Facebook dataset and tested two traversal patterns over three partitions in order to understand the effect that node order in mapped memory has on graph traversal. Due to critical errors in our dynamic partitioning feature, we are presenting results from static runs only where partitioning is performed prior to traversal. We also describe the intended tests in the dynamic environment.

All tests were run on a Windows10 machine running Oracle VM Virtual Box loaded with a Ubuntu 16.04 i386 (64-bit) image. We set the virtual machine to run on a single-core CPU with 1GB RAM. Although inspired by other related work where much of the large scale graph processing is done in a distributed or multi-processor environment, we chose to perform our experiments on a single processor both for ease of access and simplicity of analysis.

### 3.1 Space Performance

We generated graph objects from a variety of Stanford datasets when deciding which was most useful for us (see Table 1). We found significant, though not unexpected, file blow up on datasets with large N (number of vertices) and large D (max degree). Since our graph object has a neighbour-list where entries are of fixed length (smallest power of 2 greater than max degree, outlined in Figure 1), the space requirement of our file is $O(ND)$. In some cases, like that of Friendster and YouTube graphs, this was a prohibitive restriction on the department servers.

[2]Downloaded on Nov 22, 2017 from https://snap.stanford.edu/data/egonets-Facebook.html.

#### 3.1.1 Experiment

In order to put Facebook's space usage costs into context, we generated both star and complete graphs of degrees in powers of 2 (from 1 to 10). The Facebook graphs were pruned to match these degree values ($D$).

#### 3.1.2 Results

For each degree, we plotted the ratio of graph size to raw file size. Figure 2 demonstrates the behaviour of low node count with high degree (star) and high node count with high degree (complete). The star graph object waste increases linearly with degree whereas the complete graph shows a near constant ratio. The Facebook graph can be seen to behave much closer to the complete graph.

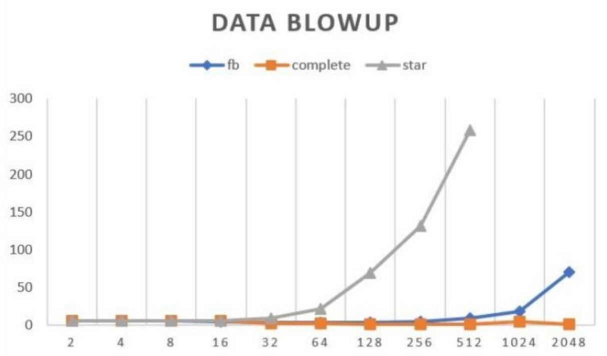


Figure 2: Graphs with few high degree nodes experience the largest space waste ratio (graph object file size to raw edge list file size).

### 3.2 Fault Performance

We measure the number of faults that occur under partitioning methods.

### 3.2.1 Experiment

We hypothesize that by having neighbouring nodes placed on the same page, we can reduce page faults, which in turn speeds up application performance. To validate this, we simulated an optimal oracle (e.g. running BFS on a BFS partitioned graph) and compare to a random ordering. We ran a 3 x 3 x 2 experiment as summarized in Table 2.

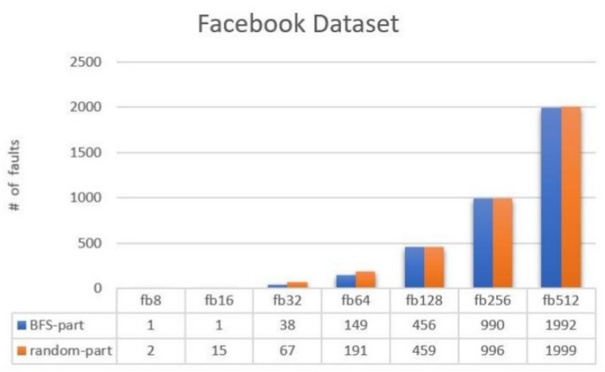
Table 2: Experimental factors

| Data (3 levels) | | Partition (3) | | Traversal (2) |
|---|---|---|---|---|
| Space inefficient (Star graph) | | BFS | | BFS |
| Space efficient (Complete graph) | x | DFS | x | |
| Real world data (Facebook dataset) | | random | | DFS |

### 3.2.2 Results

We measured the fault rate for both the star and complete graphs and found no difference in their fault rates. This is expected as only one adjacency list needs to be brought into memory for both traversals on both graphs. We report the fault rates of the Facebook graph here.

Contrary to our hypothesis, fault rate was identical for both DFS and BFS partitions. We suspect that this is due to a logical error in our DFS partitioning scheme. Printing the traversal and partitioning order showed correct traversal order; corrections will require some in-depth investigation. For our purposes here, we report these numbers as BFS partition results henceforth, and compare them to the random partitioning scheme.



| | fb8 | fb16 | fb32 | fb64 | fb128 | fb256 | fb512 |
|---|---|---|---|---|---|---|---|
| ■ BFS-part | 1 | 1 | 38 | 149 | 456 | 990 | 1992 |
| ■ random-part | 2 | 15 | 67 | 191 | 459 | 996 | 1999 |

Figure 3: fb8, fb16,.. and so on, refer to the pruned facebook datasets with max degree of 8, 16,.. respectively, plotted against the number of faults for each type of graph map partition strategies

Fault rate for partitions by traversals on the Facebook data are collected at each $D$ value. From Figure 3, we observe differences in fault count between BFS and random partitions at all $D$-degree values. Looking more closely at the fault rate per page (Figure 4, we notice this discrepancy is more apparent for smaller size of degree ($D < 128$).

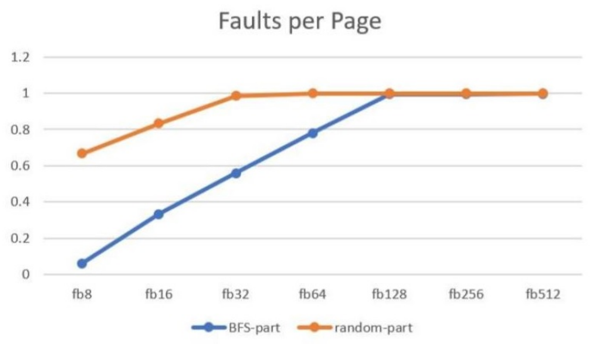


Figure 4: This figure plots the ratio of total number of faults to total number of pages for each dataset and across the different partition strategies

### 3.2.3 Dynamic Environment

Although we were unable to perform an analysis of dynamic partitioning, we expect that in terms of number of faults that it performs comparably to the static partitioning. The much more interesting evaluation would be the running time of dynamic partitioning compared to static. Our expectation is that each page fault would take longer to service as additional work needs to be performed. It is our belief that even if correct our current code would need to be heavily optimized before the extra time spent partitioning on the fly would be offset by the fewer number of faults incurred. Notably we are also only counting page faults only from the application code. The handler also incurs faults that are serviced by the kernel directly. While this increases the number of faults it also consolidates them hopefully improving performance.

## 4 Discussion

In this section, we comment on our findings and reflect on the design and implementation process, highlighting areas that we would have approached differently in hindsight.

## 4.1 Performance

We originally designed our adjacency-list to be fixed-width in order to ensure that swap was straightforward, however, this turned out to be a major barrier to good fault performance, especially in graphs that follow a power law of degree distributions —a small number of nodes have very large degree. This draw back can be clearly seen in the comparison of the star graphs (an extreme version of a power law) with the complete graph (which has a constant degree for all nodes). This could also be compared to a path graph that would also not see a large blow-up in size as it also has a constant degree per node. The Facebook graph was expected to follow a power law and exhibit poor memory usage, however this small subsection of the social network is artificially dense. Interestingly Facebook has a bound on the maximum number of friends a user may have at 5000.

Not only does this dependency on the maximum degree lead to inefficient space usage for graphs that follow a power law, but as the size of the adjacency lists grows, the number of nodes that fit on a single page decreases. Having fewer nodes per page means that there is less options for partitioning and that faults will be more common regardless of the partition. At the extreme end, this can be seen where only one node fits on a page and a fault is incurred at every access and there are no options for partitioning. This effect is visible in Figure 4 where, at smaller degrees, the breadth first partition out performs the randomized partition, but this effect disappears entirely as the degree increases.

To address the space inefficiencies of our file type we could use variable width adjacency lists however the swap function would be come considerably more complicated and likely less efficient. In light of this tradeoff, we went with the former.

## 4.2 Stumbling Blocks

Throughout the implementation phase of our project there were numerous issues that arose which represented a disproportionate amount of the time spent on this project. Most of these issues resulted from multiple format changes of the graph file coupled with a leaky abstraction existing in the pre-processing steps. This lack of modularity meant any issues or changes in the graph file structure propagated to all the pre-processing. While some of these issues should of been forseen they all represent learning opportunities and we feel we are better for having suffered through them.

### 4.2.1 Changing File Definition

When dealing with memory mapped files we initially were treating the file as an array of characters which overflowed very quickly when trying to scale to larger graphs as well as caused issues when trying to read and write to the files. This was addressed by changing to integers. Subsequently when dealing with larger graphs we were calculating offsets into the file in number of bytes, and again we overflowed these offsets when storing them as integers. Expecting a desire to accommodate especially large graphs we decided rashly to change everything to an unsigned long. It was only when calculating how many nodes actually fit on a page did we realize that none of our graphs were large enough to overflow an int and that only the offset was the issue. Realizing that unsigned longs consumed much more space than integers, giving us less freedom for re-arranging partitions, we switched nearly everything back to integers. It is only after reflecting on these issues and writing this paper that we have realized that we still did not get this quite right and should be using unsigned integers as none of the indices, degrees, nor node labels can be negative.

In hindsight we realized a lot of the complications that arose from this could have been dealt with by manipulating pointers directly rather than offsets. Specifically we should have addressed the start of the adjacency lists and and the header. This would also allow the header to be shared between both the application and the page fault handler, removing the data consistency issues that ultimately lead to this component not being completed on time.

### 4.2.2 Testing Data

While the Stanford datasets [1] were an invaluable resource, the documentation on them is rather lacking. It was only when switching to different data sets that we realized there were a number of inconsistencies within each. Specifically whether nodes start at zero or one, and if the node labels fill a contiguous range. When trying to scale our approach to larger graphs we became aware that the CS department servers have a 2Gb quota per user. The IT helpdesk was kind enough to try and track down extra swap space to be used instead, but even that was insufficient for storing the larger data sets. We summarize the datasets we tried to work with and the sizes of the resulting graph objects (see Table 1).

## 5 Related Work

To better understand the issues surrounding large-scale graph analysis and existing solutions, we began with a foray into the literature where we drew heavy inspiration. Here, we highlight the key insights that helped us gain a foothold into the field.

Locality and computational load are cited as key problems; where applications like Hadoop and MapReduce behave inefficiently, data propagation is highlighted as a key bottleneck [6, 5]. Straddling main memory and SSDs, the PrefEdge [6] system uses node access sequences to perform read-aheads to pull anticipated vertices from slower SSD to faster main memory. The design of our dynamic partitioning has parallels in doing look-ahead (in a much simpler manner), and populating a memory space to prepare for traversal.

Another graph computation strategy is to load large graphs entirely into main memory across many nodes for distributed processing. Pregel [5], as an example, is a vertex-centric bulk synchronous model that uses message passing so vertices can update their own states and that of their outgoing edges in parallel. However, many distributed graph processing systems work best when nodes are of small degree. Unfortunately, real-world graphs, as point out earlier, often follow a power law set [3]. This becomes difficult to assign across nodes as it is highly skewed and thus unbalanced by nature.

For the purposes of this paper, we acknowledge the space challenges and explore how memory mapping on a single local machine could improve locality. Lin et al (2014) defined a graph storage format for memory mapped (mmap) computations that use an "index file" to track node page locations using offsets and an "edge list file" that defines the graph [4]. Our format has similarities in that our header stores node offsets. For graph definition, however, we elected to borrow from the vertex-centric model and aggregate connected nodes in adjacency lists. This enables us to incorporate the flexibility of look-aheads and swapping yet retain consistency in vertex manipulation.

These papers helped us determine a graph representation, which turned out to be arguably the most time-consuming and lesson-laden exercises —designing the graph API, determining a possible structure for our custom graph object, and exploring how to read and write from mapped memory.

## 6 Conclusion

While we fell short of our implementation goals in building graph partitioning on-the-fly, we exceeded our individual learning goals for this project. We are more comfortable with C and gained an OS level understanding of paging and memory maps, enough to understand APIs like userfaultfd and accompanying machinery such as ioctl. We also discovered tools to debug and monitor system memory performance, chiefly to investigate how an application is using the memory subsystem, including the type and amount of memory used by a process, where it is being allocated, and how effectively the processor's cache is used. We also picked up some fun hacks like alternative ways to clear both disk and processor caches. In addition to learning about specific features of the Linux environment, we also gained valuable experience in debugging systems code, working cooperatively as software development team, and how to handle failures during the development process.

## 7 Acknowledgements

We would like to thank our classmates for consulting and helping us evolve our project ideas. We are particularly indebted to Amanda Carbonari and Tony Mason for patiently answering our frantic questions about userfaultfd, Surbhi Palande for her optimistic encouragement and guidance in developing the project, and the rest of the members of the NSS lab for welcoming us into their work-space. And finally, to Dr. Andy Warfield for directing us through the course, his numerous insights, and inspiring personality.

## References

[1] Stanford large network dataset collection. `https://snap.stanford.edu/data/`. Accessed: 2017-12-12.

[2] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz. Recent advances in graph partitioning. In *Algorithm Engineering*, pages 117–158. Springer, 2016.

[3] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, volume 12, page 2, 2012.

[4] Z. Lin, M. Kahng, K. M. Sabrin, D. H. P. Chau, H. Lee, and U. Kang. Mmap: Fast billion-scale

graph computation on a pc via memory mapping. In *Big Data (Big Data), 2014 IEEE International Conference on*, pages 159–164. IEEE, 2014.

[5] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[6] K. Nilakant, V. Dalibard, A. Roy, and E. Yoneki. Prefedge: Ssd prefetcher for large-scale graph traversal. In *Proceedings of International Conference on Systems and Storage*, pages 1–12. ACM, 2014.