SPEEDING-UP GRAPH PROCESSING ON SHARED-MEMORY PLATFORMS BY OPTIMIZING SCHEDULING AND COMPUTE

BY

AZIN HEIDARSHENAS

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois Urbana-Champaign, 2021

Urbana, Illinois

Doctoral Committee:

Professor Josep Torrellas, Chair
Professor Deming Chen
Professor Wen-mei Hwu
Professor Rakesh Kumar
Assistant Professor Sasa Misailovic
Assistant Professor Adam Morrison, Tel-Aviv University

ABSTRACT

Graph processing workloads are being widely used in many domains such as computational biology, social network analysis, and financial analysis. As DRAM technology scales down into higher densities, shared-memory platforms gain increasing importance in handling large graph sizes.

We study two main categories of graph algorithms from an implementation perspective. Topology-driven algorithms process all vertices of the graph at each iteration, while data-driven algorithms only process those vertices that make a substantial contribution to the output. Furthermore, the performance of a graph algorithm execution can be broken down into three components, namely, pre-processing, compute, and scheduling. For data-driven algorithms, the work of each thread is driven by the dependencies between vertex values that are known only at run-time. Hence, the scheduling will take a significant portion of execution. However, for topology-driven algorithms, the scheduling time is negligible since the work of each thread can be determined at compile-time.

In this dissertation, we present three techniques to address the performance bottlenecks in both data-driven and topology-driven algorithms. First, we present Snug, which is a chip-level architecture that mitigates the trade-off between synchronization and wasted work in data-driven algorithms. Second, we present V-Combiner, which is a software-only technique to mitigate the trade-off between performance and accuracy of topology-driven algorithms using novel vertex-merging and recovery mechanisms. Finally, we present KeepCompressed, which is a set of algorithms to speed-up compute for topology-driven algorithms using vertex clustering for dynamic graphs.

To my family, for their love and support.

ACKNOWLEDGMENTS

I would like to express my deepest appreciation to my advisor, Dr. Josep Torrellas, without whose unwavering support and guidance I could not have imagined to thrive during my Ph.D. journey. I am deeply thankful for the endless hours he spent on brainstorming ideas to continuously expose me to exciting new research directions and opportunities. I would like to extend my sincere thanks to my committee member/collaborator, Dr. Sasa Misailovic, who has been a crucial asset in shaping major ideas proposed in this dissertation. Sasa has been always available for brainstorming discussions, constantly motivating me to push for my ideas, even during the hardest times. Additionally, I am deeply indebted to my committee member/collaborator, Dr. Adam Morrison, from whom I have learned extensive knowledge in the field of parallel and distributed processing. Adam always generously spent hours during our meetings, proposing ideas for improving this dissertation, despite a huge time-zone difference.

I am extremely grateful to my other committee members: Dr. Wen-mei Hwu, Dr. Deming Chen, and Dr. Rakesh Kumar for their invaluable insights and suggestions to improve the quality of this dissertation.

I very much appreciate my colleagues and fellow lab members Serif Yesil and Dimitrios Skarlatos for their crucial assistance in preparing the papers during submission periods as well as their constant availabilty for problem-solving discussions. Special thanks goes to my colleague Tanmay Gangwani, who has been a true professional and a great mentor in my first research project. I would like to also acknowledge the assistance of my other lab members Bhargava Gopireddy, Raghavendra Pradyumna Pothukuchi, Mengjia Yan, Yasser Shalabi, Thomas Shull, Jiho Choi, Andy Gong, Antonio Franques, Apostolos Kokolis, and Houxiang Ji, during seminars and discussions.

Last but not least, I would like to thank my lovely husband Mohammad, my wonderful parents, and my dear brother for their unconditional love and support during the most difficult times.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Problem Definition

Many parallel high-performance parallel computing applications such as those in the domains of machine learning, social network, computational biology and financial analysis operate on graphs. Each graph connects different entities based on their relationships. As graphs become larger and more complex, processing them requires even larger computational and memory resources.

As the DRAM technology scales down to achieve higher density, shared-memory platforms gain increasing importance in handling large graph sizes. New servers can support up to 1 TB of DRAM. Moreover, in most of the graph applications, the output of the algorithm is defined per vertex. Therefore, the amount of memory required by the application is proportional to the number of graph vertices. The amount of memory to store the graph data structure is also proportional to the number of edges. The largest graph dataset available is the hyperlink graph [1] with 3.5 billions of vertices and 128 billions of edges. Therefore, the total amount of memory required by a typical graph application such as page rank is around 493 GB. This is much less than the DRAM capacity current large servers can support today [2].

We study two main categories of graph algorithms from an implementation perspective [3]. To compute the values for all the vertices, a graph operator is applied iteratively to the each vertex. *Topology-driven* algorithms process all vertices of the graph at every iteration, even if they have negligible contribution to the output of the application. These algorithms are implemented in a synchronous fashion, such that all threads have to synchronize before moving to the next iteration. In contrast, *data-driven* algorithms start with processing all the vertices but later stop processing those vertices which will no longer have any contribution. In this model, the computation is driven by

the dependencies between neighboring vertex values at run-time. Therefore, data-driven algorithms are more efficient if implemented in an asynchronous fashion, effectively having different threads process different iterations of the graph algorithm at the same time.

The performance of a graph algorithm execution can be broken down into several categories. First, there is *pre-processing* time, which is the time spent building the graph data structure. The rest of the execution is the time spent running the graph algorithm which itself can be broken down into 1) *compute* and 2) *scheduling*. The compute time is the time spent doing the actual computation using the graph algorithm operator. The scheduling time is the time spent balancing out and distributing the work across threads. In topology-driven algorithms, the work can be distributed among the threads before the execution with the help of a compiler. Therefore, the scheduling time is negligible and most of the algorithm time is taken by the compute. However, in data-driven algorithms, the work that each thread performs is not known prior to the execution since it is driven by the dependencies between neighboring vertices at run-time. Therefore, the work should be divided among the threads dynamically at run-time. For example, all threads need to access a shared queue that maintains a collection of work items. In this setting, threads need to be synchronized when accessing the shared queue. Unlike topology-driven algorithms, scheduling time takes significant amount of the overall execution time in data-driven algorithms.

Finally, many real-world graphs are dynamic and may change over time. New vertices or edges may be added or some of the old vertices or edges may be removed, which we refer to as *graph updates*. For such graphs, the *pre-processing* time include *update* time.

1.2 Dissertation Contributions and Roadmap

The goal of this dissertation is to optimize the performance bottlenecks of graph processing workloads on *shared memory platforms* by optimizing scheduling and compute times, for both static and dynamic graphs. We make a key observation that, using graph summarization and compression techniques, we can omit the unnecessary computation in many graph processing workloads and hence speed-up their execution. Specifically, we propose a

set of pre-processing compression algorithms and show that their implementation on shared-memory platforms can be light-weight. The result is faster overall execution, compared to the OpenMP baselines implemented using the GAP benchmark suite framework [4]. The baselines are hand-optimized by tuning for the best OpenMP dynamic scheduling granularity.

We present three techniques to mitigate the performance bottlenecks in both data-driven and topology-driven algorithms. The chapters of this dissertation are organized as follows:

- In Chapter 2, we present Snug [5], a chip-level architecture, that aims to speed-up the scheduling in data-driven algorithms and mitigates the trade-off between synchronization and work-efficiency of the parallel data-driven graph algorithms.

- In Chapter 3, we introduce V-Combiner [6], a software-only technique that speeds-up the compute in topology-driven algorithms and mitigates the trade-off between performance and accuracy of the graph algorithm.

- In Chapter 4, we present our work KeepCompressed that proposes a set of algorithms to speed-up compute in topology-driven algorithms for dynamic graphs.

- Finally, in Chapter 5, we conclude the dissertation and outline future directions for the extension of our work on speeding-up graph processing workloads using graph compression techniques.

CHAPTER 2

SNUG: ARCHITECTURAL SUPPORT FOR RELAXED CONCURRENT PRIORITY QUEUEING IN CHIP MULTIPROCESSORS

2.1 Introduction

Many parallel algorithms in domains such as graph analytics [7, 8, 9], discrete event simulation [10], and machine learning [11] rely on *priority-based* task scheduling. When the algorithm creates a task T, it assigns it a priority p, and if $T_1.p < T_2.p$, then T_1 should execute before T_2. (That is, lower p values mean higher priorities.) The data structure commonly used to implement priority-based task scheduling is the *concurrent priority queue* (PQ). A PQ maintains a collection of items, each associated with a priority. It supports two basic operations: *enqueue*, which adds an item to the correct position in the queue, and *dequeue*, which removes the item with the highest priority from the head of the queue.

Unfortunately, concurrent PQs are not scalable. Since every thread executing dequeue tries to remove the same highest-priority item, the head becomes a synchronization hotspot [12, 13, 14, 15, 16, 17]. The resulting serialization and synchronization overheads can dominate execution time.

To avoid this problem, researchers have proposed to *relax* PQ semantics and allow dequeue to return an item that is not the highest-priority one [18, 19, 20, 21]. Relaxed PQ algorithms use various strategies to find the item to dequeue. For example, they perform a short random walk on a skip list to find an item to dequeue [18], or pick the highest-priority item between a thread-local PQ and a global shared PQ [20]. These strategies alleviate the bottleneck at the head.

Relaxed PQ algorithms face the danger of straying too much from the desired task execution order and ending-up performing *wasted work*. For

example, in a discrete event simulation, a thread may process an event that is far in the future, only to have to reprocess it again later, as new events that occur from now until that time change the system state. Thus, the key difficulty in designing a relaxed PQ is to consistently return high-priority items.

In practice, existing relaxed PQs often fail to achieve this goal. For example, the SprayList [18] returns an item among the first $O(t \log^3 t)$ ones in the PQ with high probability, where t is the number of threads. For a 64-thread execution, this translates to a weak guarantee of returning an item within the first $13,824$ in the queue (ignoring constant factors). Similarly, with the recommended parameter $k = 256$, the k-LSM PQ [20] only guarantees returning an item within the first $16,384$ in the queue in a 64-thread execution. Current priority-based task scheduling algorithms thus pose a synchronization vs. work-efficiency tradeoff: alleviating the synchronization hotspot by using relaxed PQs leads to wasted work.

This chapter introduces novel hardware support to address this tradeoff. Our chip-level architecture, called Snug, relaxes the priority order in a PQ algorithm slightly, to alleviate contention while inducing, on average, little wasted work. Snug distributes the PQ into subqueues, and maintains a set of *work registers* that point to the highest-priority task in each subqueue. A new *PickHead* instruction returns a high-priority task from the combined PQ.

PickHead employs an adaptive technique to pick a high-priority task without congesting the network. Sometimes, PickHead reads all work registers in parallel, saves the result, and returns a random task from within the R highest-priority ones observed. R is called the *relaxation count*, and is dynamically adapted by Snug, based on the rate of synchronization failures—which indicate the degree of contention. In later PickHead invocations, PickHead reuses the saved information to avoid rereading the work registers. Finally, PickHead sometimes reads only from a set of local work registers, to save traffic. Overall, PickHead performs high-quality task selection while avoiding hotspots, minimizing wasted work, and consuming acceptable network bandwidth.

Snug Effectiveness. We evaluate Snug on a simulated 64-core chip using graph and discrete event simulation applications with various inputs. We compare the execution time of the applications using Snug and using

several other PQ algorithms. Such algorithms include software-only PQs based on a concurrent skip list, SprayList, and distributed skip list; and a hardware-based centralized PQ. Snug reduces the average execution time of the applications by 1.4×, 2.4× and 3.6× compared to state-of-the-art concurrent skip list, SprayList, and software-distributed PQs, respectively. Compared to the latter two relaxed PQ designs, Snug reduces the number of wasted tasks by 3.3× and 36.8×, respectively.

Contributions. We make the following contributions:

• The Snug architecture, which consists of the PickHead instruction, the work registers, and other ISA and microarchitectural extensions. This design adapts its degree of relaxation dynamically.

• Simulation-based evaluation of Snug using graph and discrete event simulation applications, and a comparison to several other software- and hardware-based PQs.

2.2 Motivation and Background

We target a group of parallel algorithms that benefit from *priority-based scheduling*. Such algorithms decompose work into tasks (which can generate new tasks as they run) and execute them in order according to some notion of priority. Processing a task out of priority order leads to *wasted work*, where a task executes inefficiently and/or the results of its computation are thrown away later.

2.2.1 Need for Priority-Based Task Scheduling

Priority-based scheduling has use cases for parallel algorithms in a wide range of domains. To name some examples, in discrete event simulation, Time Warp [22, 10] optimistically executes simulated events (i.e., tasks) in parallel, rolling back events that are discovered to have been simulated before their dependencies are satisfied. Priority-based scheduling minimizes such rollbacks. In machine learning, task execution order significantly impacts convergence time in residual belief propagation [23, 11], an algorithm for performing inference on graphical models. Parallel algorithms for important graph problems, such as single-source shortest path (SSSP) [7, 8], minimal

spanning tree (MST) [24, 9], and betweenness centrality [25], also leverage priority-based scheduling.

Here, we use SSSP as a running example. SSSP is a classic graph problem that is used in many domains. SSSP is also a building block for computing betweenness centrality which, in turn, has wide application in network theory, social networks [26], and biology [27].

The SSSP Example. Given a weighted directed graph and a *source* node s, SSSP finds the weight of the shortest path from s to every other node in the graph. Most SSSP algorithms use *relaxations* [28], in which the algorithm tests whether a shortest path found so far can be improved. Each node v is associated with a $dist(v)$ label (initially 0 for s and ∞ for all other nodes). A relaxation considers an edge (u, v) of weight w. If $dist(v) > dist(u) + w$, the algorithm updates $dist(v)$ to be $dist(u) + w$. Updates of a node's label are synchronized (e.g., with atomic instructions), allowing relaxations to run in parallel. The algorithm thus converges to the labels containing the weights of the shortest paths from s.

Any relaxation that does not update a node's label to its true distance is *wasted work*, as it will be overwritten by the relaxation that updates the label to the true distance. Dijkstra's SSSP algorithm [28, 7] relaxes each edge exactly once. It partitions the graph into *explored* nodes, whose distance from s is known, and unexplored nodes. In each iteration, it picks the unexplored node u with the smallest label, marks it explored, and relaxes every edge (u, v).

Priority-based Task Scheduling. By using node labels as *priorities* and defining a task as relaxing every outgoing edge of a node, we can approximate Dijkstra's algorithm in a multiprocessor and minimize wasted work. Note that due to parallelism, wasted work is not guaranteed to disappear. This is because two tasks may perform conflicting relaxations.

2.2.2 Concurrent Priority Queue Algorithms

Concurrent Priority Queues (PQs) are natural data structures to implement priority-based task scheduling. A PQ maintains a collection of items (i.e., tasks or nodes), each associated with its own priority p. We consider lower p values to be higher priorities. A PQ supports two operations: *enqueue*,

which adds an item to the collection in its correct position, and *dequeue*, which removes the item with the highest priority from the collection and returns it.

PQs suffer from scalability problems, due to synchronization hotspots resulting from the fact that every thread executing dequeue tries to remove the same, highest-priority item. Motivated by this scalability problem, researchers have proposed *relaxed* PQs, which allow a dequeue to return a task that is not the highest-priority one [18, 19, 20, 21]. Relaxed PQs alleviate the synchronization hotspot, but increase the amount of wasted work. As a result, the performance of the application may improve or degrade with a relaxed PQ.

Contention in Standard PQs. Modern PQs [12, 29, 14, 16, 17] are implemented based on the *skip list* data structure [30]. A skip list is conceptually a sorted linked list in which some nodes contain "hints" that enable searching in logarithmic time. PQs implement enqueue by inserting an item into the skip list (which is sorted by priority), and dequeue by removing the head item.

For simplicity, we explain the PQ synchronization issues using the simpler *linked list*-based PQ, rather than the skip list-based PQ, and cover skip lists in the Appendix. Linked-list insertions are performed by searching the sorted list for the correct place to insert the new item, and linking a new node there. Searching is done using only reads, without acquiring locks [31], and so enqueue operations can proceed in parallel and scale well.

Linked-list deletions are done in two phases [31, 32]: the node is first logically deleted by setting a "deleted" flag in the node, and then physically deleted by updating the next pointer of its predecessor. We explain the details in the Appendix. Both of these steps involve synchronization, either through locks or atomic Compare-And-Swap (CAS) instructions. Crucially, threads concurrently performing a dequeue all attempt to delete the same node, resulting in a significant synchronization bottleneck. PQ research has focused on reducing the impact of this bottleneck, e.g., by batching physical deletions [14]—but inevitably hits a scalability limit.

Relaxed PQs. Relaxed PQs eliminate the bottleneck at the PQ head by distributing dequeue operations, often using randomization. The SprayList [18] is based on a skip list, but a dequeue chooses its target item by perform-

ing a short random walk on the list. MultiQueues [19] and Sheaps [33, 34] distribute the data structure, composing the logical PQ out of per-thread PQs. Dequeues are performed from a remote queue only if the local one is empty [34], or from a queue chosen randomly [19]. Other approaches combine per-thread PQs with a global PQ [20].

Relaxed PQ designs trade off synchronization costs with increased probability of wasting work, since the tasks returned by a relaxed PQ may be far from the highest-priority one.

2.3 The SNUG Architecture

We first explain the key idea of Snug and introduce its architecture. We introduce the proposed instructions and their interface to the software as well as enqueue and dequeue operations in the later subsections.

2.3.1 Main Idea

Our goal is to design architectural support for a PQ that minimizes both wasted work and synchronization overhead. We accomplish it with Snug, which is a novel chip-level architecture for high-performance, distributed PQs. Snug exists in the context of a directory-based manycore. Each core (or core cluster) owns a module of the distributed directory. Any programmer-declared logical queue is distributed into multiple physical queues. Each physical queue's head is in a different directory module.

When a thread running on a core calls the enqueue operation, the PQ library leverages the Snug hardware to enqueue the node at the correct spot in the core's local physical queue. When the thread calls the dequeue operation, the PQ library leverages the Snug support to return to the thread a node with one of the highest priorities in the distributed PQ. As we will see, the hardware uses a special algorithm, so as to minimize both wasted work and synchronization overhead, and avoid excessive traffic. Overall, with Snug, enqueues are fast because they are local, while dequeues are both fast and return high-priority nodes thanks to special hardware.

Figure 2.1 shows the Snug architecture. In a tiled manycore, each node has two hardware structures: a set of *work registers* in the directory module,

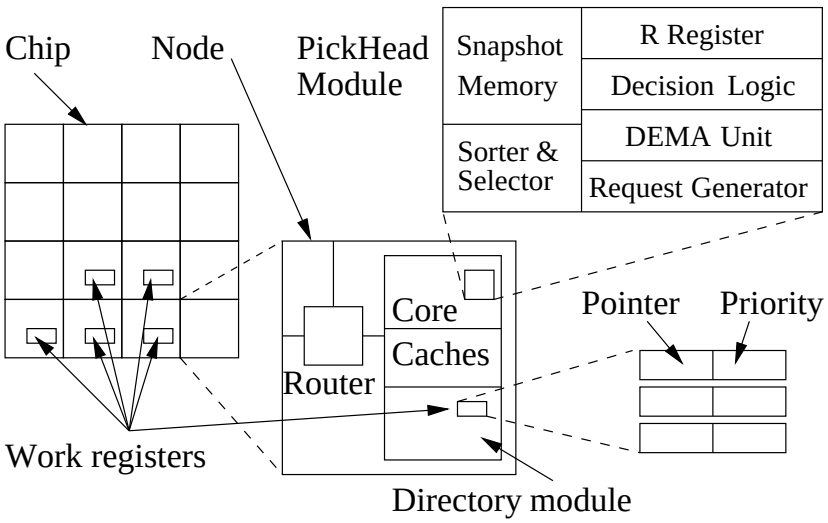


Figure 2.1: SNUG architecture in a directory-based manycore. Each directory module has a set of work registers, and each core has a PickHead module.

and a *PickHead* module in the core. Each work register can function as the head of a work queue. Cores can access all the work registers in the chip as memory-mapped locations in an uncacheable virtual address range. When a work register is used, it has two values: a pointer to the first node in the corresponding queue, and that node's priority. Any core in the chip can read a *s* without causing ping-ponging of cached data across the network. In addition, there are instructions that read both pointer and priority together, and that modify both pointer and priority atomically.

When a thread calls the dequeue operation, the PQ library issues a new instruction called PickHead. The instruction may access multiple queues, only the local queue, or no queue at all. For each of the queues accessed, the network transaction returns the information in the work register.

The PickHead module in Figure 2.1 supports the PickHead instruction. When the instruction visits multiple queues, it issues parallel requests that read each of the relevant work registers in parallel. The values returned (pointer to the node at the head and its priority) are stored in a small *snapshot memory* in the module. Then, PickHead does *not* return to the software the pointer to the very highest-priority node in the snapshot memory. Doing so would cause all the threads to attempt a CAS on the same node. Instead, the SNUG hardware sorts the pointers to the nodes in the snapshot memory

based on their priority, considers only the top R of them, and picks one of them at random to return to the software. The software will then attempt to perform a CAS on the node.

R is the *relaxation* count. It is stored in the R register of the PickHead module (Figure 2.1) and is dynamically adapted by SNUG based on the rate of synchronization failures.

Sometimes, PickHead chooses to visit no queue. It simply reuses information in the snapshot memory, returning another of the node pointers there. This choice saves traffic, and is selected a few times right after the snapshot memory is refreshed. However, using this choice several times in a row risks using stale data, which will result in the failure of the subsequent CAS operation.

Finally, PickHead sometimes chooses to visit only the local queue. This choice triggers a cheap transaction, which induces minimal traffic and contention. However, it may or may not provide a pointer to a high-priority node to dequeue. This choice is selected several times after multiple entries from the snapshot memory have been used, to reduce traffic pressure.

After PickHead returns a pointer to a node, the library attempts to dequeue the node with a CAS. If it fails, it calls PickHead again, and the process repeats until a node is successfully dequeued.

2.3.2 PickHead Instuction and Module

The goal of the PickHead instruction is to return a pointer to one of the highest-priority nodes in the PQ with minimal overhead. The instruction uses the PickHead module shown in Figure 2.2. As the instruction starts executing, a decision logic module makes one of three choices: issue a global access to multiple work registers, issue a local access, or issue no network access at all.

Global Access. Under certain conditions, the PickHead instruction issues as many network requests as there are physical queues in the requested logical queue. These requests proceed in parallel, each visiting a queue and returning a 3-tuple to the snapshot memory and sorter & selector module (Figure 2.2). A tuple has the queue ID, i.e., the virtual address (VA) of the queue, and the two contents of its work register. Now, the PickHead module needs to pick

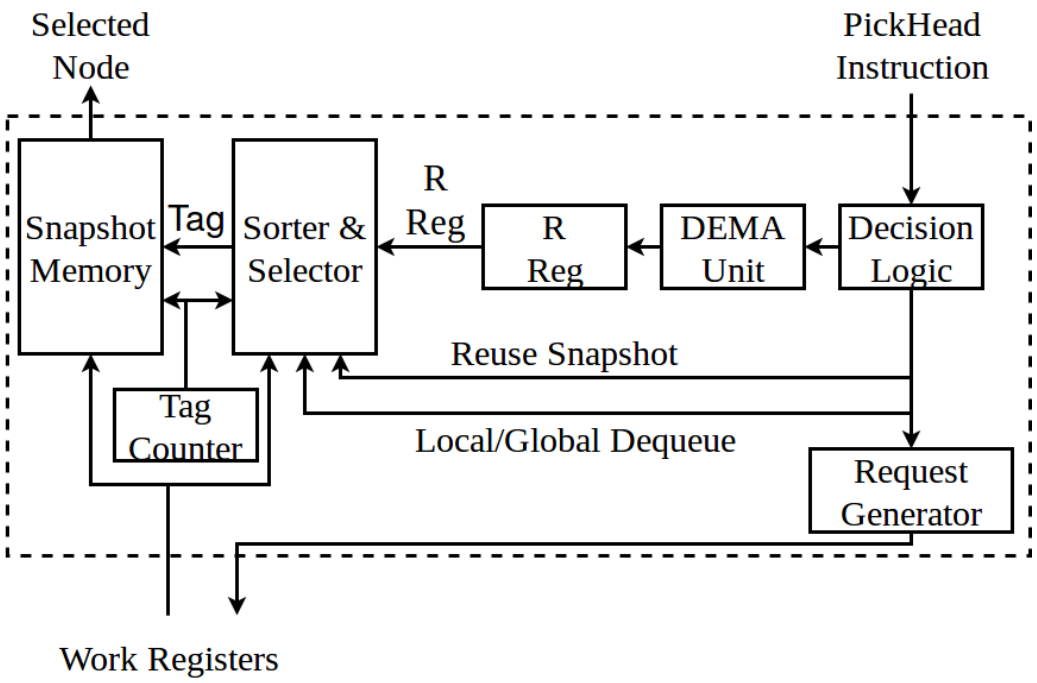


Figure 2.2: PickHead module.

one of these node pointers to return to the software, which will then perform a CAS to attempt to dequeue the node from the corresponding queue.

As indicated above, if the chosen node was always the highest-priority one, we would induce high PQ contention. Hence, the sorter & selector module (Figure 2.2) sorts the node pointers as they arrive in decreasing priority, and then considers the subset of them that have the highest priorities. This subset is called the *inclusion set*. Its size is equal to the relaxation count stored in the R register (Figure 2.2). Finally, the hardware randomly selects one of the node pointers in the inclusion set and returns it to the software. Section 2.4.2 describes this step.

To set R, we would ideally use feedback from the contention for the PQ and the amount of wasted work performed by the application. Unfortunately, wasted work is very application dependent, and programmers may find it difficult to estimate. On the other hand, PQ contention is easy to measure. Hence, in Snug, the PickHead instruction takes an operand (*PrevFailed?*) that is set if the CAS on the node provided by PickHead in the previous global access failed. This operand is set and reset inside the PQ dequeue library. In general, if the frequency of failures of CAS operations is high,

we increase R; if it is low, we decrease R. The Double Exponential Moving Average (DEMA) unit (Figure 2.2) uses the CAS success/failure history to set R. Section 2.4.3 explains the algorithm used.

Local Access. In this case, PickHead accesses only the local queue. This operation is inexpensive, but can potentially obtain a pointer to a node with a lower priority than if we performed a global access. The returning data bypasses the Snapshot memory and sorter & selector, and is returned to the software.

In networks organized in clusters, this transaction could involve PickHead accessing all the queues in the local cluster. If so, this case would proceed as explained for the global access: the multiple responses would be received in the snapshot memory and sorted, and one node pointer from the highest-priority ones would be returned to the software.

To balance the desire to reduce the global traffic and to pick high-priority nodes, we design PickHead to perform a fixed number of local accesses between any pair of consecutive global accesses. Section 2.4.1 explains the algorithm we used.

No Network Access. After a PickHead performs a global access, several subsequent PickHead executions reuse the information in the snapshot memory. The goal is to reduce the network traffic without hurting the quality of node selection much. Specifically, a prior global access brought node pointers from all the queues, sorted them, picked one, and consumed the corresponding node. Now, the PickHead instruction reuses the sorted array of pointers in the sorter & selector module as follows. The hardware re-considers all the non-consumed entries in the sorter & sorter module, and randomly selects one to return to the software. That entry is then marked as consumed. Note that the algorithm is otherwise not selective in what nodes to re-consider—this is to maximize the chances of attaining an available node. This process is repeated a few times before the snapshot is discarded. Section 2.4.1 presents the algorithm.

2.3.3 Interface to the Software

The Snug hardware is accessible with the API in Table 2.1. The table lists the inputs and outputs of each of the four Snug operations. While different

implementations are possible, we envision *PickHead*, *UpdateHead*, and *FetchHead* to be actual instructions, with their operands placed in registers—some encoded in the instruction, and some used implicitly. The other operation, *AllocHeads*, is a subroutine that invokes the memory manager.

AllocHeads allocates work registers. It takes two inputs and one output. The inputs are the number of programmer-visible logical queues to allocate, and an array with the number of physical queues that each logical queue should have. The physical queues of each logical queue are distributed into as many different directory modules as possible. AllocHeads returns the virtual addresses (VAs) of all the physical queues allocated, i.e., the VAs of all the work registers allocated.

Table 2.1: API of the Snug hardware.

AllocHeads input: # of logical queues input: Array with the # of physical queues for each logical queue output: VAs of all the physical queues allocated
PickHead input: ID of the logical queue input: Did the CAS after previous global access to the queue fail? output: VA of the chosen physical queue to dequeue from output: Pointer to the node at the head of the chosen queue
UpdateHead input: VA of the physical queue input: Pointer to the node at the head of the physical queue input: Pointer to the node to place at the queue's head input: Priority of the node to place at the queue's head output: Successful or failed outcome
FetchHead input: VA of the physical queue output: Pointer to the node at the head of the physical queue

PickHead was described in Section 2.3.2. It takes as inputs the ID of a logical queue and a boolean that indicates if the CAS on the node provided by PickHead in the previous global access to the queue failed. The outputs are the VA of the chosen physical queue to dequeue from, and a pointer to the node at the head of that queue. In the worst case, the latency of this instruction is that of multiple uncached accesses in parallel to directory modules, plus sorting the incoming node priorities in hardware.

```
1  void enqueue(int LogQueID, Node *new) {
2      Node **head = &queue[LogQueID][local]; // local queue
3      // head is the uncacheable address of the head of the queue
4      while (true) {
5          Node **prev = head;
6          Node *curr = FetchHead (head); // get node at head of queue
7          while (new->priority > curr->priority) { //high num is low prio
8              prev = &curr->next;
9              curr = curr->next;
10         }
11         // insert node between prev and curr
12         new->next = curr;
13         if (prev == head)
14             ok = UpdateHead (head, curr, new, new->priority);
15         else
16             ok = CAS(prev, curr, new);
17         if (ok)
18             return; // success
19 }   }
```

(a) Enqueue

```
20 PrevFailed? = no; // local variable, initially false
21 void* dequeue(int LogQueID) {
22     // "first" will get information on the node to dequeue:
23     // VA of its physical queue, and a pointer to the node
24     struct {
25         Node **physqueue;
26         Node* head;
27     } first ;
28     while (true) {
29         first = PickHead (LogQueID,PrevFailed?); // return node
30         if (first.head == NULL)
31             return NULL; // empty
32         Node *next = first.head->next;
33         success = UpdateHead (first.physqueue, first.head, next, next->priority);
34         if (Global)
35             PrevFailed? = (success ? no : yes);
36         if (success)
37             return first.head;
38 }   }
```

(b) Dequeue

Figure 2.3: Code to enqueue (a) and dequeue (b) a node with SNUG. Recall that enqueues are always local.

UpdateHead performs a CAS to modify a work register and, hence, change the head of the corresponding physical queue. It changes the two fields of the work register, i.e., the pointer to the head node and its priority—atomically. It is used in both the enqueue and dequeue PQ library operations. UpdateHead takes four inputs (Table 2.1): the VA of the physical queue, the expected value of the pointer to the node at the head of the queue, a pointer to the node that the instruction wants to place at the head of the queue, and that node's priority. If UpdateHead succeeds, both pointer and priority in the work register are updated; if it fails, no change is made. UpdateHead

returns a boolean with the outcome of the operation.

This operation is performed in a CAS hardware unit in the directory module. In this way, operands do not have to flow from the directory module to the core and back. A similar unit is provided in current GPUs to perform CASes in the cache hierarchy [35, 36]. Also note that, with this instruction, SNUG can perform arbitrary writes to the queue head. Overall, the overhead of this instruction is an uncached access to a directory module that includes a CAS operation.

FetchHead reads the pointer field of a work register and, thus, obtains the head of a physical queue. It is used as part of the enqueue library, which needs to check the current value of the queue head to be able to change it with UpdateHead. FetchHead takes as input the VA of the physical queue. Its output is a pointer to the node at the head of the queue. Its overhead is an uncached access to a directory module.

2.3.4 Enqueue and Dequeue Operations

The previous instructions are not typically used directly by programmers. Instead, they are used in the PQ library to allocate queues, and to enqueue and dequeue nodes. Figures 2.3 (a) and (b) show pseudo-code for the routines that enqueue and dequeue a node, respectively. Programs that call the PQ library do not know about physical queues; they reference logical queues.

For simplicity, Figures 2.3 (a) and (b) focus on the physical deletions. We omit the details of handling logical deletions (Section 2.2.2), which are orthogonal to SNUG. In addition, the figures use simple linked lists. In practice, high-performance concurrent PQ libraries use the more efficient skip list [16].

The enqueue routine in Figure 2.3 (a) is called with the ID of a logical queue and a node to enqueue. The routine first determines the VA of the local physical queue (Line 2); this is the queue where the node will be enqueued. The routine then uses FetchHead to read the pointer to the node at the head of the queue (Line 6). It then follows the linked list of nodes, reading the priority of each node, to find the place to insert the new node (Lines 7-10). If the node needs to be placed at the head, it uses UpdateHead to do so and fill the work register (Line 14). Otherwise, it uses a plain CAS (Line 16). Either

UpdateHead or CAS can fail; if it does, the routine goes back to walking the queue to find where to enqueue.

The dequeue routine in Figure 2.3 (b) is called with the ID of a logical queue. It returns one of the highest-priority nodes from the logical queue (not necessarily the highest one), by dequeueing it from the appropriate physical queue. The routine uses PickHead to find the VA of the physical queue and a pointer to the node (Line 29). Then, the routine tries to dequeue the node using UpdateHead on the appropriate physical queue (Line 33). If this was a global access, the routine sets *PrevFailed?* based on whether UpdateHead succeeded or failed (Line 35). *PrevFailed?* will be used the next time that PickHead performs a global access. In any case, if UpdateHead succeeds, the routine returns a pointer to the dequeued node (Line 37). Otherwise, the routine repeats the use of PickHead and UpdateHead until the dequeue succeeds or there is no node to dequeue.

2.4 Detailed Aspects of SNUG Design

In this section, we describe detailed algorithms of the Snug architecture. We first explain the algorithm for picking the nodes and further describe how to sort the nodes and set the relaxation count in the later subsections.

2.4.1 Algorithm to Pick the Node to Process

Snug's algorithm for picking the next node to dequeue maintains a balance between two objectives. On one hand, Snug wants to observe globally up-to-date information about the PQ, so that it can pick a high-priority node and minimize wasted work. On the other hand, Snug also wants to avoid generating excessive network traffic, which would be the case if all PickHead invocations visited all the work registers. As per Section 2.3.2, Snug achieves this balance by complementing global accesses with (i) reuses of snapshot memory information, and (ii) accesses to the local queue.

The decision logic unit in the PickHead module (Figure 2.2) divides the stream of PickHead instruction executions into *phases*. Each phase begins with a PickHead instruction that performs a global access, followed by U

PickHead executions that reuse the snapshot, and finally L PickHead executions that access the local queue.

A global access reads many pointers to nodes, each of which is at the head of a physical queue. One of these pointers is returned to the software. However, many of the pointers read by the global access are good candidates for processing. Hence, in the next U calls to PickHead, we want it to return some of these nodes. Eventually, the snapshot data becomes stale, as other cores have dequeued nodes from the various queues. Therefore, after U PickHead executions, SNUG discards the snapshot.

SNUG then switches to visiting only the local queue for L times. This decision trades off the quality of the executed work for short periods, in order to avoid excessive network traffic. If the network is organized in clusters, each of these L PickHead executions could access all of the local queues in the cluster, and return one of the best nodes in them. In any case, following these L local accesses, a new phase begins, and the next PickHead triggers a global access.

2.4.2 Sorting the Nodes

The sorter & selector module sorts the 3-tuple responses as they arrive from memory. We use an inexpensive sorter that performs *pipelined linear insertion sort* (Figure 2.4). As each of the 3-tuples arrives, a tag counter (Figure 2.2) assigns it a tag. Then, the incoming priority and tag are fed to the sorter & selector module, where the earlier arrivals are already sorted in registers. The incoming priority is then compared to the existing priorities in sequence, until the hardware determines its correct position (Figure 2.4). At that point, the incoming priority and tag are stored in the corresponding register, and all remaining existing priorities are shifted one position to the right. Details of a similar system can be found in [37].

The hardware complexity of this module and the sorting time scale linearly with the number of physical queues. Specifically, for n queues, the sorting network has n comparators. When the last tuple is received, it takes n steps to complete the sorting. Our analysis with design tools estimates that each step takes 1 ns, or 2 processor cycles. Hence, sorting takes $2n$ processor cycles after the last tuple arrives.

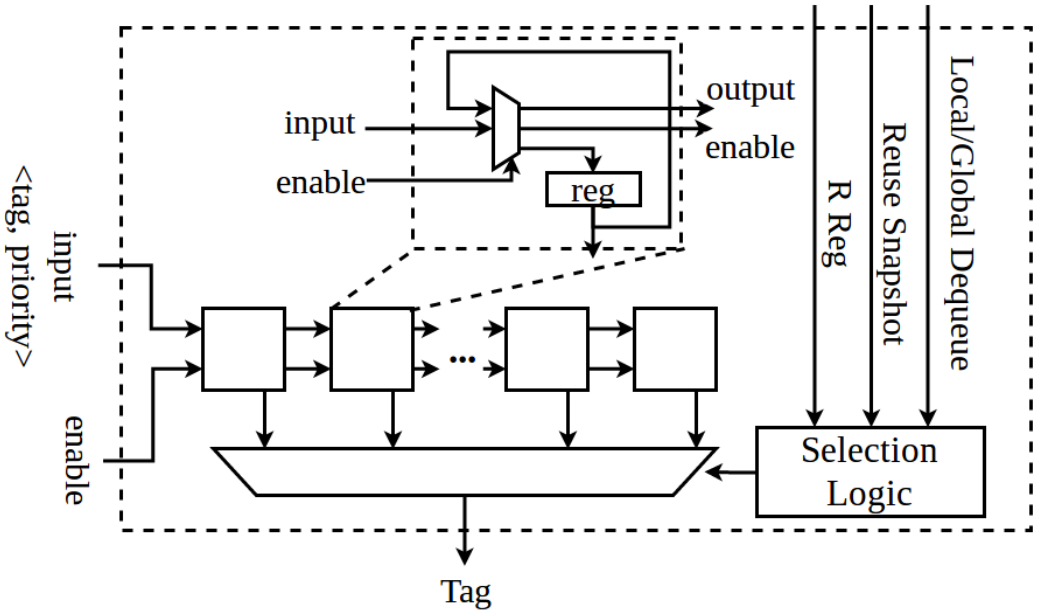


Figure 2.4: Sorter & selector module.

After the sorting, some selection logic reads a register that contains the current time and performs a modulo operation of the time with the current value of R. The result is an index that the module uses to select one of the top R entries from the sorted array of priorities (Figure 2.4). The chosen tag is then passed to the snapshot memory, and used to read the corresponding physical queue ID and node pointer, which are then passed to the software. The sorted list of priorities is kept latched in the sorter & selector module for reuse by future PickHeads.

2.4.3 Setting the Relaxation Count (R)

The relaxation count (R) is a parameter that determines the quality of the nodes obtained by the global accesses and the no-network accesses. It is defined as the maximum number of sorted nodes in the sorter & selector module from which one will be returned to the software. If R is too high, it may induce wasted work; if it is too low, it may cause UpdateHead failures in the dequeue routine in Figure 2.3 (b) because many cores will collide. To set R, we use real-time feedback from the application on how frequently these UpdateHead operations fail. Since only a PickHead that performs a

global access creates a fresh snapshot, we use the failure rate of only the UpdateHead operation immediately following a global access, to decide how to change R. The value of R serves as an indicator of the global contention in the system.

While the optimal R is likely to change across the execution time of an application, it is important that its value be impervious to short-term fluctuations of UpdateHead failure rate, and instead reflect long-term trends. For this reason, we use the Double Exponential Moving Average (DEMA), which is based on the Exponential Moving Average (EMA) [38], a widely used indicator in statistical technical analysis. DEMA smoothing is preferred over EMA smoothing, especially when the series exhibits some trends.

As shown in Figure 2.2, the PickHead module includes a DEMA Unit, which receives information from the Decision Logic on how frequently UpdateHead failures occur for a global access. The information is a single bit: 1 for failure, 0 for success. Intuitively, observing many occurrences of 1 suggests that R is too small, while observing many 0 values suggests it is too high.

We define a *segment* as a series of bits of the same value (either 1 or 0) passed by the Decision Logic to the DEMA unit. On a segment termination, the DEMA unit uses the length of the segment to compute the following:

$$EMA = \alpha * sign * length + (1 - \alpha) * EMA$$
$$DEMA = \beta * EMA + (1 - \beta) * DEMA$$

where *length* is the number of entries in the segment, and α and β are constants. The variable *sign* is 1 for a segment of 1s and -1 for a segment of 0s. With this setup, UpdateHead failures tend to push the DEMA slowly in the positive direction, while a string of UpdateHead successes move it in the opposite direction. Note that the DEMA changes only slowly.

When the application starts, the DEMA unit uses a default initial value R_0. During execution, the unit divides the time into *windows* of fixed size, and keeps calculating the DEMA. It keeps a positive DEMA threshold (T_{pos}) and a negative one (T_{neg}). If the DEMA has been above T_{pos} anytime in each of the previous two windows, the DEMA unit increases R one notch; if the DEMA has been below T_{neg} anytime in each of the previous two windows, R decreases by one notch. Figure 2.5 shows an example of this algorithm.

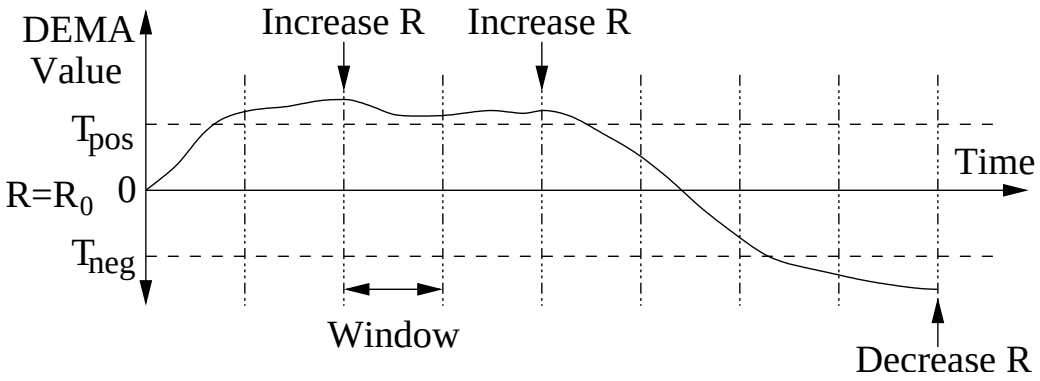


Figure 2.5: Example of how SNUG sets R.

Since each core computes its DEMA value independently, each core modulates its own R independently. Interestingly, relaxation by one core has the serendipitous effect of reducing contention for sibling cores. As such, in most scenarios, some cores converge on smaller R values than others.

R changes as the application executes different sections, and as different applications execute. However, this process is invisible to the programmer. After each context switch, R is reset to R_0.

2.4.4 Multi-Socket Configuration

SNUG does not have any centralized hardware and relies on hardware cache coherence. Consequently, it can also be implemented in a cache-coherent multi-socket system. In such a system, the PickHead requests may need to obtain the data from remote sockets, like regular loads. The software needs no changes.

2.5 Evaluation Environment

In the following subsections, we describe the evaluation methodology of SNUG. We first describe the modeled architecture, and introduce the evaluated concurrent PQs and parallel applications in the later subsections.

2.5.1 Architecture Modeled

We perform our evaluation with cycle-level simulation of a 64-core chip using the gem5 simulator [39]. Table 2.2 shows the baseline architecture modeled. We model a hierarchical network with clustering: every eight cores share a single L2, and the eight cache-coherent L2s are connected to a shared, 8-banked L3 with a crossbar.

Table 2.2: Parameters of the architecture evaluated.

Parameter	Value
Architecture	64 cores on chip, 2GHz cores
Private L1 Caches	32KB-I, 32KB-D WB, 8-way, 2 cycles hit latency
Per-cluster L2 Cache	1MB WB, 16-way, 12 cycles hit latency
Shared L3	16MB WB, 8 banks, 16-way, 30 cycles hit latency
Cache line size	64B
Coherence	Two-level MOESI directory protocol
Network	32B wide, hierarchical, crossbar with snoop filters
Main memory	$\approx$ 200 cycles
SNUG Parameters	
Reuses (U); Local acc. (L)	4; 4
R_0; DEMA Window	32; 100K cycles
DEMA thresholds/constants	T_{pos} = 5, T_{neg} = -2.5, α = 0.6, β = 0.6
Sorting network delay	64 ns = 128 cycles

The latency of the new instructions is modeled as follows. FetchHead is an uncached access to one directory module. UpdateHead is like FetchHead plus four cycles for a CAS. PickHead's latency depends on the operation: in a global access, it is n parallel uncached accesses to directories, plus $2n$ cycles to sort the incoming messages (Section 2.4.2) plus four cycles to return the node pointer to the instruction; in a local access, it is one uncached accesses to a directory plus four cycles to return the node pointer to the instruction; in a no-network access, it is four cycles to return the node pointer to the instruction.

AllocHeads is invoked before the section of the application that is timed.

2.5.2 Concurrent PQs Compared

We compare the following concurrent PQs:

Concurrent Skip List (*SW-SK*). This is a skip list implementation [14] where the levels for new skipnodes are chosen based on a geometric distribution. The maximum number of levels is 24.

Concurrent Spraylist (*SW-SP*). This SprayList builds on top of the skip list by spraying the pops over a range of starting nodes in the list. We use the SprayList parameters as advised by the authors in [18]. The spray is started at the height of $\lfloor \log_2 t \rfloor + 1$, and the jump length at each level is $\lfloor \log_2 t \rfloor + 1$, where t is the number of threads. The number of levels to descend between jumps is 1, and the maximum number of levels is 24.

Distributed Software (*SW-D*). This is a software distributed PQ with a per-core skip list. Threads always enqueue to their local skip list. For dequeue, the local skip list is tried first. If the queue is empty, the thread attempts to steal work from nearby skip lists.

Centralized Hardware (*HW-C*). This is a centralized version of Snug. It consists of a single shared skip list with a single, centralized work register. It uses FetchHead and UpdateHead to access the work register. A single work register obviates the need for PickHead.

Distributed Hardware (*HW-D*). This is Snug. It uses per-core skip lists, each of which is supported by a work register at the directory. Threads always enqueue to their local queue. For dequeue, a PickHead instruction returns the node to dequeue as described in Section 2.4.1. For the L local accesses, PickHead visits the local queue. However, if the data in the Snapshot memory indicates that the local queue is empty, PickHead instead randomly visits one of the queues that the snapshot memory indicates are not empty. Table 2.2 lists Snug's parameters.

Software Version of Snug. In this case, a global dequeue scans in software all the heads of the distributed queues. This PQ is an order of magnitude slower than SW-D, due to the overhead of serially scanning all the queue heads. Therefore, we omit it from the evaluation.

Omitted PQs. We have also evaluated the OBIM priority-based scheduler from Galois [8]. OBIM trades off synchronization time in exchange for executing tasks out of priority order and potentially performing unnecessary work. OBIM has been tuned for multi-socket platforms, which have

sizable synchronization costs. For the manycore architecture considered in this chapter, with a single chip and very low synchronization costs, we find that OBIM sometimes performs substantially worse than most of the other PQ implementations—even after careful parameter tuning. Hence, since we are unable to tune OBIM well for our hardware, we do not include it in the evaluation.

We also do not include a comparison to the k-LSM PQ [20]. k-LSM is a two-level software PQ where the access frequency to the top-level PQ is controlled by the k parameter. As k grows, fewer accesses go to the top-level queues and the quality of returned tasks decreases. We do not present the k-LSM PQ because we found that on our inputs, SW-SK is on average 1.45× faster than k-LSM (even for the best value of k we found, which is $k = 256$) in experiments on a real machine with 64 threads. We remark that prior work [40] only evaluated the k-LSM PQ on a random graph input, whereas we use more realistic inputs, as described below.

2.5.3 Applications Evaluated

We execute four applications with a variety of inputs. The applications come from graph analytics (SSSP, BFS, and A*), and from an event-driven simulation model (SIMUL).

Graph Analytics. SSSP uses Dijkstra's algorithm [28] to compute the shortest distance to all graph nodes, starting from a source node (Section 2.2.1). We base our implementation on the push-operator [8], since we found it to outperform the pull-operator based approach. breadth-first search (BFS) uses breadth-first search in graphs where the weight of all the edges is 1, which drastically changes the task scheduling behavior compared to SSSP. A-star (A*) is a pathfinding algorithm used to compute the shortest distance from a source to a target node. It relaxes only a subset of the graph nodes, and uses heuristics to guide the searching process.

Event-driven Simulation. SIMUL is a system-modeling program with a variety of use cases [41, 42]. We model the execution of discrete events, some of which have dependencies on other events. It uses a BFS-style graph traversal on the dependency graph. When a thread dequeues an event, it checks if all the event's dependencies have been executed. If true, the event executes and the dependents are pushed to the PQ; otherwise the event is

ignored. For inputs, we generate a matrix of dependencies between events based on the approach outlined in [18]. The number of events that are dependent on a given event is parameterized with δ, and the mean distance between the source and dependent node is parameterized with γ.

Table 2.3 shows the inputs evaluated. By default, runs are with 64 threads. The distributed PQs use 64 queues, one local to each core. HW-C, which is Snug, uses 64 work registers.

Table 2.3: Program inputs.

Input	Description
Graphs for SSSP and BFS:	
NY Road (NY)	Road network for New York City with edge weights as the travel time [43]. $\|V\|$ = 264,346, $\|E\|$ = 733,846
Amazon (Ama)	Amazon product co-purchasing network from the SNAP dataset [44]. $\|V\|$ = 262,111, $\|E\|$ = 1,234,877
Scalefree (Sf)	Graph with power law degree distribution, built with R-MAT [45, 8]. $\|V\|$ = 260,237, $\|E\|$ = 2,097,152
Matrices for SIMUL:	
M_1	Number of events = 12K, $\delta = 15$, $\gamma = 100$
M_2	Number of events = 12K, $\delta = 15$, $\gamma = 200$
Grid for A*:	
TaleofTwoCities (T2C)	2D grid map from Starcraft benchmark set [46]

In the applications, a thread repeatedly dequeues a task from the PQ, processes it, and (possibly) enqueues one or more tasks to the PQ. We categorize the work into *good work* and *wasted work*. For SSSP, BFS, and A*, good work consists of graph edge relaxations that update the label of a node to its true distance (Section 2.2.1), and wasted work consists of all the remaining, redundant relaxations. Wasted work occurs either due to thread concurrency or, most notably, relaxed PQ semantics. For SSSP and BFS, the number of tasks performing good work remains constant across the different PQ designs since all the graph nodes are reduced to their shortest distance. In contrast, A* terminates when the target node is relaxed to its shortest distance. Since different PQs explore the grid in different ways, the number of tasks performing good work can change with different PQs. For SIMUL, good work consists of dequeueing an event after all of its dependencies have executed, thereby enabling its own execution. Otherwise, the task is classified as wasted. The number of tasks performing good work remains constant across the different PQ designs.

2.6 Evaluation

In this section, we demonstrate the results of evaluating Snug architecture. Subsection 2.6.1 compares the execution times of the applications and inputs for different PQ implementations. Subsection 2.6.2 compares the network traffic of different PQ implementations. Subsection 2.6.3 provides an analysis of wasted work of the applications and inputs for different PQ implementations. Subsection 2.6.4 demonstrates Snug scalability on larger architectures. Subsections 2.6.5 and 2.6.6 study the sensitivity of the applications to Snug parameters and characterize Snug adaptivity to synchronization failures. Finally, Subsection 2.6.7 provides an analysis of area and power of the Snug architecture.

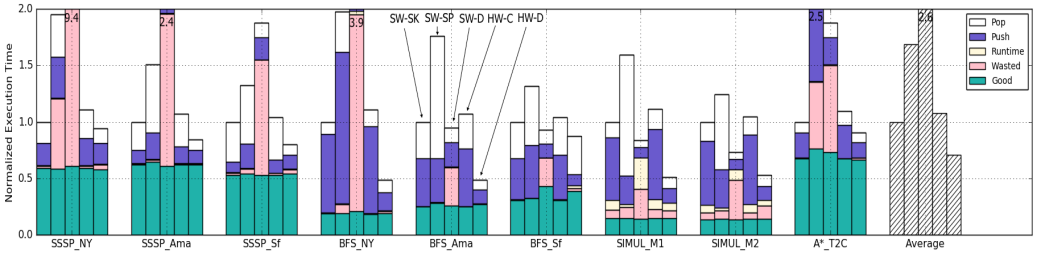


Figure 2.6: 64-core execution time of the evaluated applications and inputs on different priority queue implementations.

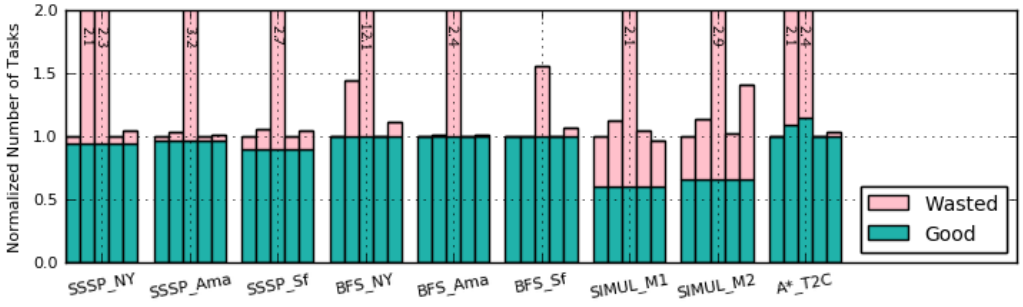


Figure 2.7: Breakdown of the tasks in the applications into good and wasted tasks. The bars in each application and input are ordered as in Figure 2.6.

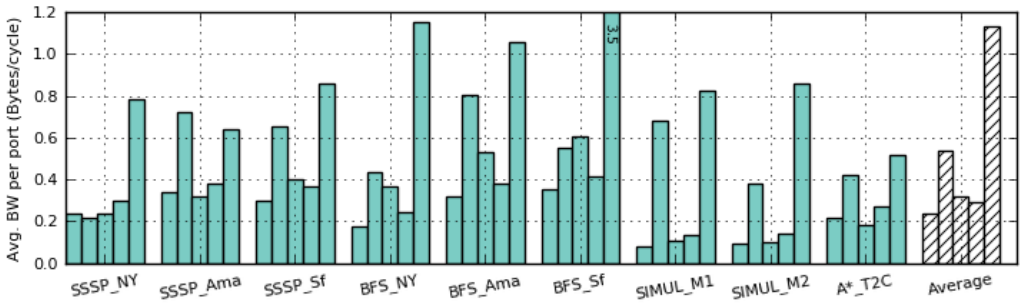


Figure 2.8: Average bandwidth consumption in each of the global crossbar ports connected to the core clusters. The bars in each application and input are ordered as in Figure 2.6.

2.6.1 Execution Time

Figure 2.6 compares the execution time of the applications and inputs under the different PQ implementations. For each application and input, we show, from left to right, SW-SK, SW-SP, SW-D, HW-C, and HW-C, all normalized to SW-SK. The time is broken down into *pop* synchronization (the time spent in PQ dequeue operations), *push* synchronization (the time spent in PQ enqueue operations), runtime (the execution of auxiliary code, such as thread termination), wasted work, and good work.

To help understand these bars, Figure 2.7 provides a breakdown of the total tasks executed in the program, classified into good and wasted tasks. The total number of tasks is normalized to the number in SW-SK. The bars are ordered as in Figure 2.6. Note that wasted tasks affect the execution in two ways. First, they induce redundant processing of graph edges or events. These cycles appear as wasted work in Figure 2.6. Second, they cause extraneous PQ enqueues and dequeues, hence increasing push and pop times in Figure 2.6.

In the following, we discuss all the PQ implementations in detail except for HW-C. HW-C differs from SW-SK mostly by having a single work register, pointing to the highest-priority task. HW-C's overall performance is similar to SW-SK. On average, it performs slightly worse than SW-SK due to the extra overheads of the new instructions. Recall that HW-D is Snug.

SSSP. As shown in Figure 2.6, HW-D performs better than the other PQs. On average across the three inputs, it reduces the execution time by 1.2×, 1.8×, and 5.1× compared to SW-SK, SW-SP, and SW-D, respectively. Com-

pared to SW-SK, HW-D reduces the pop time substantially, while only increasing the push time slightly. HW-D reduces pop because it eliminates the contention on the queue head with the use of distributed queues and the PickHead instruction, avoiding contention. HW-D's push time is slightly higher than SW-SK because HW-D is more relaxed, and ends up creating slightly more bad work (Figure 2.7) and enqueuing more tasks.

HW-D performs much better than SW-SP and SW-D, as SW-SP suffers from pop time, and SW-D from wasted work. SW-SP does not work well for the relatively short PQs in these applications. SW-SP returns tasks from among the first $O(t \log^3 t)$ ones in the PQ (with high probability), where t is the number of threads (Section 2.2.1). SW-SP's random walks often reach the end of list; when this happens, SW-SP performs a linear scan of the PQ, to make sure no work was missed [18], and this increases pop time. In addition, in sparse graphs like NY-Road, the short PQ causes SW-SP to return a random task, ignoring priorities, and causing wasted work.

SW-D has significant wasted work for all three graphs. This is because, in this relaxed PQ, each thread takes tasks from its own queue, without looking for the highest-priority task. The lack of synchronization overheads in SW-D is unable to compensate for this wasted work.

BFS. HW-D also performs well in BFS, reducing the average execution time by 1.7×, 3.1×, and 3.7× compared to SW-SK, SW-SP, and SW-D, respectively (Figure 2.6). In BFS, all the edges of the graph have a weight of one. This increases the available parallelism, since there are now multiple paths of the same distance from the source node to each destination. This changes the behavior of the PQ implementations.

HW-D attains better performance than SW-SK and SW-SP because of reduced synchronization. The push time is very high for these PQs, because many tasks now have the same priority and so get pushed to the same region of the skip list—leading to contention. HW-D reduces the push time because enqueues do not contend with each other and, in addition, each enqueue traverses a shorter local queue.

SW-D has substantial wasted work. The reason is that each thread dequeues tasks from its local queue, and so is less likely to find the highest-priority tasks. HW-D eliminates most of the wasted work because dequeuing with PickHead enables a global view of available tasks, and provides higher-priority tasks. Note that on average, the wasted work with SW-D in BFS is

lower than in SSSP. This is because in BFS, many more tasks have the same priority, and so the local queue is more likely to have one of the globally-highest priority tasks.

SIMUL. HW-D reduces the average execution time by 1.9×, 2.7×, and 1.5× compared to SW-SK, SW-SP, and SW-D, respectively. The SW-SK and SW-SP PQ implementations have significant synchronization overheads. Specifically, SW-SK has push contention similar to BFS, and SW-SP has high pop time due to the previously-discussed issue of a sub-optimal spray. SW-D is the best software alternative. It reduces the push and pop times by using local enqueues and dequeues. Local dequeues, however, lead to many task dependency violations—a task is dequeued for execution before its dependencies have executed. This causes wasted work with SW-D (Figure 2.7). HW-D has lower contention overheads compared to the SW-SK and SW-SP PQs, and less task dependency violations compared to SW-D.

A*. HW-D also performs best, reducing the execution time by 1.1×, 2.8×, and 2.1× compared to SW-SK, SW-SP, and SW-D, respectively. In A*, the priority with which a task is inserted in the PQ includes an extra heuristic cost, which is the Euclidean distance to the target. We see that SW-SP and SW-D have significant wasted work due to the relaxation of nodes away from the optimal path from source to target. SW-SK performs worse than HW-D due to higher push time.

Overall. The data illustrates the main PQ tradeoff: the SW-SK and SW-SP PQs suffer from push and pop contention, while the SW-D PQ suffers from wasted work due to bad task selection. HW-D addresses both problems. Averaged across all applications and inputs, HW-D in Figure 2.6 reduces the execution time by 1.4×, 2.4×, and 3.6× compared to state-of-the-art skip list, SprayList, and software distributed PQs, respectively. These are substantial execution time reductions, as they correspond to 64-core executions. Compared to the latter two PQ designs, HW-D reduces the number of wasted tasks by 3.3× and 36.8×, respectively.

2.6.2 Network Traffic

To assess Snug's network traffic, Figure 2.8 shows the the average bandwidth consumption in the ports of the global crossbar that connect to the core clusters, in bytes per cycle.

We see that HW-D consumes more bandwidth than the other concurrent PQs. The average increase ranges from 2.2× relative to SW-SP, to 5.6× relative to SW-SK. However, HW-D's bandwidth consumption is modest in absolute terms. On average across all applications, Snug consumes 1.13 bytes per cycle, or 3.5% of the crossbar's 32 bytes per cycle link capacity, which is the same link capacity assumed in related work [47] and deployed in Intel's Haswell.

Table 2.4 examines the behavior of the PickHead instruction accesses, averaged over all the cores. *GA* is the percentage of PickHead accesses that are global. We can see that, typically, about 1 in 6 accesses are global. TS is the average number of instructions per task. We see that this number is about 1300-4400. Next, we show the failure rate of PickHead accesses, broken down by access type. GF, RF, and LF are the failure rates for global, reuse, and local accesses, respectively. On average, we observe that local accesses have the least failure rate, followed by global and reuse. Reuse accesses fail often.

Table 2.4: PickHead accesses in HW-D and task size.

	SSSP			BFS			SIMUL		A*
	NY	Ama	Sf	NY	Ama	Sf	M1	M2	T2C
GA	19%	15%	21%	18%	15%	43%	16%	19%	18%
TS	2717	3993	4438	1297	1633	2144	2017	1895	3750
GF	48%	28%	45%	38%	32%	92%	35%	30%	21%
RF	73%	80%	71%	76%	81%	75%	55%	56%	74%
LF	11%	8%	13%	12%	14%	5%	7%	8%	5%

2.6.3 Analysis of Wasted Work

Figure 2.6 showed that SW-D suffers from substantial wasted work in SSSP and, for the NY input, in BFS. To understand why, recall that a thread in SW-D dequeues tasks from its local queue. If it obtains tasks with priority far from the globally-highest priority, SW-D will have wasted work. However, if the application is such that there is a large number of tasks for each priority, then there is a greater chance that the locally-best task has a priority similar to the globally-best priority. In that case, SW-D will have much less wasted work.

Figure 2.9 explains the different behavior of the applications and input data sets. For SSSP and BFS, it shows the number of nodes in the graph

that are at a certain distance from the source node (i.e., their priority). Figures 2.9 (a) and 2.9 (b) show the Sf graph (SSSP_Sf and BFS_Sf). We see that the distribution of priorities is different. In SSSP_Sf, the edge weights yield 250 priorities, most with only few nodes. In BFS_Sf, there are only a handful of priorities, each with tens of thousands of nodes. Hence, SW-D has less wasted work in BFS_Sf than in SSSP_Sf.

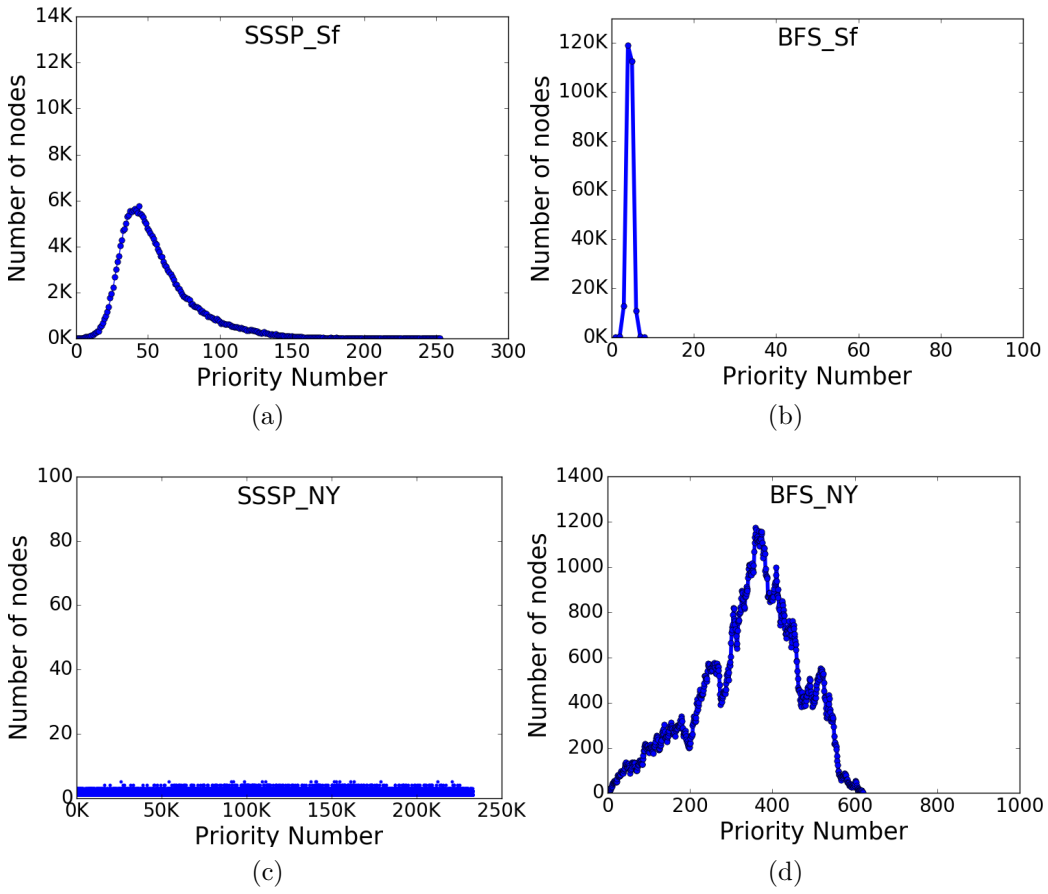
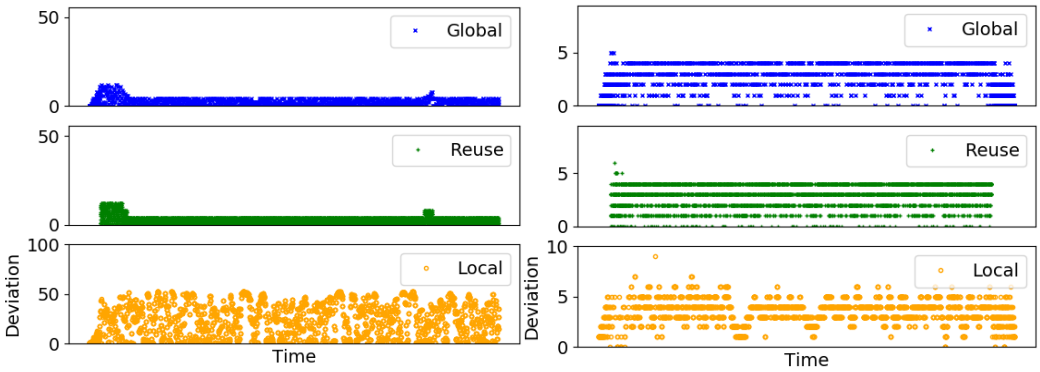


Figure 2.9: Priority distribution for applications and inputs.

In contrast, Figures 2.9 (c) and 2.9 (d) show the NY graph (SSSP_NY and BFS_NY). There are few nodes for each priority—especially in SSSP. SW-D has a high chance of picking tasks with priority far from the globally-highest priority. As a result, SW-D has wasted work in SSSP_NY and BFS_NY. In HW-D, the PickHead instruction obtains a global view of the tasks in all the queues. Thanks to this, HW-D picks high quality tasks, and has negligible wasted work.



(a) SSSP_NY (b) BFS_NY

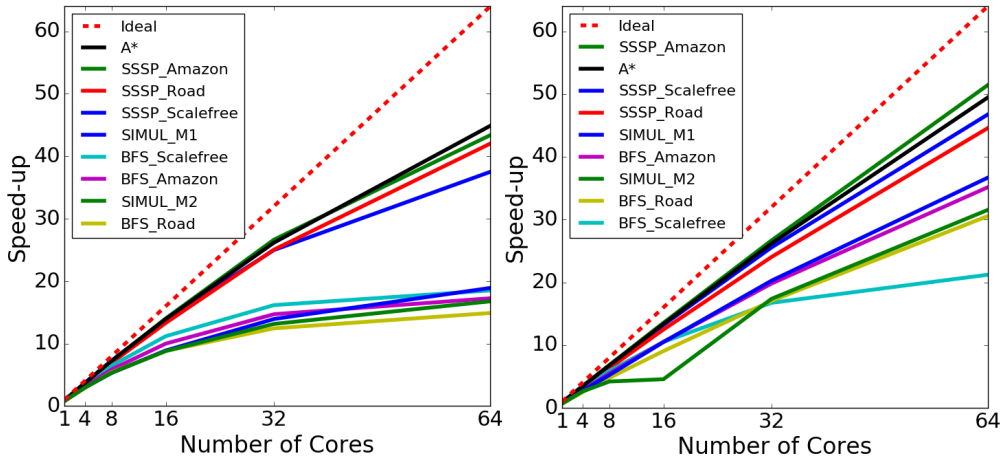
Figure 2.10: Difference between the priority returned by PickHead and the best priority in snapshot.

Figure 2.10 visualizes the degree of priority relaxation by the PickHead instruction. Specifically, we plot the difference between the priority returned by PickHead and the best priority in the snapshot, across time. The figure shows data for SSSP and BFS, on the NY graph, and for global, reuse, and local accesses. We see that the only accesses that get tasks far from the best are local accesses in SSSP. Local accesses greedily exploit the local queue and can dequeue low-quality tasks.

2.6.4 SNUG Scalability

To understand the scalability of Snug with the number of cores, Figure 2.11 shows the speedups of our applications and inputs as we change the core count from 1 to 64. Figures 2.11 (a) and Figure 2.11 (b) show data for our baseline SW-SK and HW-D, respectively. In both figures, the speedup are relative to a 1-core sequential skip-list PQ design which, unlike 1-core SW-SK, does not use CAS instructions. Figure 2.11 (a) shows that SW-SK does not scale for about half of the configurations, namely the BFS and SIMUL applications. These applications push many tasks with identical priorities, leading to contention in skip list nodes, which leads to high push times (Figure 2.6). In contrast, Figure 2.11 (b) shows that the speedup in HW-D scale better for most configurations. The distributed queues in HW-D mitigate the push contention in BFS and SIMUL (Figure 2.6), as different

tasks are pushed to different queues. Hence, HW-D is more scalable.



(a) SW-SK (b) HW-D

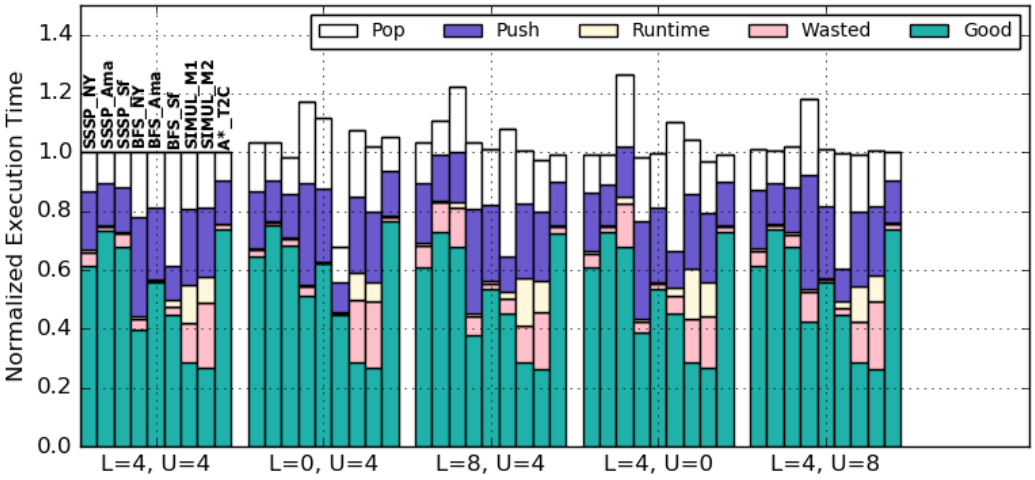
Figure 2.11: Speedup of SW-SK (a) and HW-D (b).

2.6.5 Sensitivity Analysis

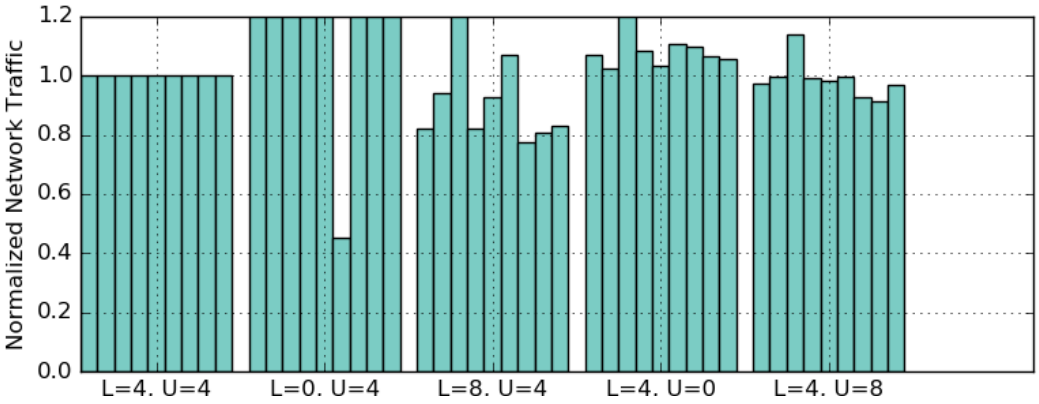
Sensitivity to L and U. SNUG's algorithm for picking the next node to process (Section 2.4.1) has two parameters: following a global access, there are U snapshot reuses and L local accesses. HW-D sets both parameters to 4. We do sensitivity analysis varying L and U, and measure the execution time and the network traffic. Specifically, we fix U to 4 and change L to 0 and 8. Then, we fix L to 4 and change U to 0 and 8.

Figure 2.12 (a) shows the execution time, and Figure 2.12 (b) the network traffic, as we vary L and U from 0 to 4 and 8. Specifically, the first set of bars is HW-D (L=4, U=4). In the next two sets of bars, U is fixed to 4 and L varies to 0 and 8. In the final two sets of bars, L is fixed to 4 and U varies to 0 and 8. For each (L, U) setting, we have a bar for each application and input. For a given application and input, the bars are normalized to the (L=4, U=4) setting.

We see that changing L alters the behavior of the system. When L=0, SNUG dequeues better tasks, by issuing more global accesses. However, the execution time does not generally go down because the traffic increases. At the other end, when L=8, SNUG dequeues more low-priority tasks from the



(a) Execution time



(b) Network traffic. Values for truncated bars are 2.3, 1.8, 1.3, 2.7, 2.0, 1.6, 1.9, 2.2, 1.8, and 2.0, from left to right.

Figure 2.12: Sensitivity to the number of local accesses (L) and reuses of a snapshot (U).

local queue. While the traffic generally decreases, the execution time does not go down because there is more wasted work. Overall, L=4 is slightly better.

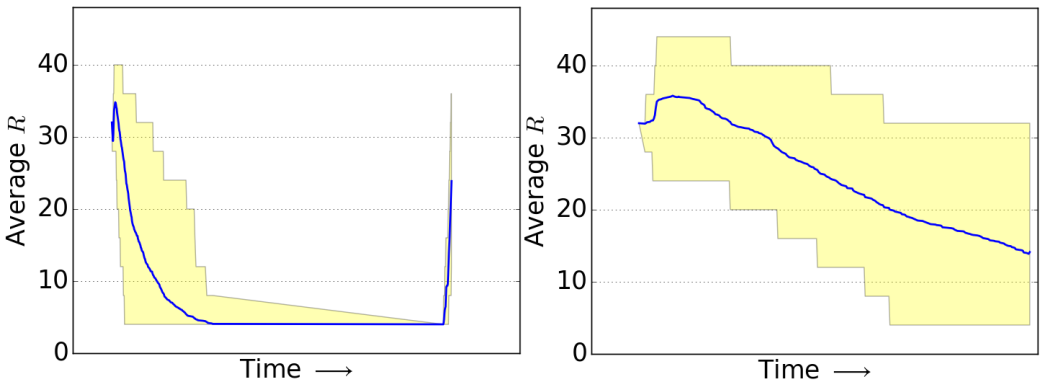
In the final two sets of bars, we change U. When U=0, the traffic and contention is higher because there is no snapshot reuse. This results in an increase in the execution time. With U=8, the execution time also goes slightly up, likely due to the fact that the snapshot is reused past the point when it is useful. In summary, U=4 is a good design point.

Sensitivity to application input. The characteristics of distributed PQs and how SNUG interacts with them could change appreciably depending on the size of input graphs or matrices. Although the computational costs

for cycle-level gem5 simulation of 64 cores prohibit us from using datasets much larger than those in Table 3, we measure their impact indirectly by downsizing the L1, L2, and L3 caches instead. Generally, we find that SNUG does not unreasonably exploit any effects due to caching of inputs. For example, for BFS_NY, we reduce all the caches to half (or a quarter) of their original capacities. We find that SNUG is 2.13× (or 2.08× for the quarter cache) faster than SW-SK, compared to 2.05× in the unmodified architecture.

2.6.6 Characterizing R Adaptivity

The PickHead module in each core adjusts the relaxation count R based on the rate of synchronization failures, which indicate the degree of contention. Figure 2.13 shows two representative examples of R's behavior as the application executes: SSSP_NY and SIMUL_M2. In the figure, the X-axis represents time. At each point in time, the figures show the average value of R across all cores. The shaded area around the mean shows the minimum and maximum values across cores. In both cases, we start with our default $R_0 = 32$.



(a) SSSP_NY (b) SIMUL_M2

Figure 2.13: Average value of R as execution progresses.

SSSP_NY demonstrates how SNUG automatically adjusts R. At the beginning of the application, there are few tasks and, as processors contend for them, R goes up. Gradually, the queues obtain tasks. Dequeuing occurs concurrently across many queues, and synchronizations succeed. This triggers a decrease in R. The descent continues as long as the cores do not observe enough dequeue conflicts. Finally, as the application runs out of work and

most queues become empty again, the contention increases, and R has a final spike.

SIMUL_M2 in Figure 2.13 (b) shows a different behavior. There is an initial hump in R as in SSSP_NY. However, the rate of descent of R is slow. The reason is that SIMUL_M2 has a smaller average task size than SSSP_NY (Table 2.4). This increases the frequency of CASes attempting to dequeue the same task. The result is higher CAS failures. SNUG adapts to this contention by keeping R sufficiently high to avoid hotspots, while dequeuing high-quality tasks.

2.6.7 Area and Power Analysis

We estimate the area and power overhead of the PickHead module by considering the overhead of the sorter & selector and the snapshot memory units. We use the Synopsys DC with the 32 nm technology library for logic and CACTI [48] with the 32 nm ITRS-hp technology for memory components. For the design in this chapter, the snapshot memory holds 64 entries, and each entry is 16 bytes (head pointer and queue ID). It is implemented as a single-ported RAM structure. The sorter & selector is composed of 64 registers and 64 comparators, to process 70-bit data (priority and tag).

Our estimates show that the area of the PickHead module is about 0.23 mm^2, and its power consumption 61.1 mW in 32 nm. To put these numbers in context, we compare to existing processors. Publicly-available information indicates that the core area (processing unit, L1, and L2) of a Sandy Bridge processor and a Power7+ processor at 32 nm are 18.18 mm^2 and 21.14 mm^2, respectively. Using data from [49], we estimate that the per-core power of a Sandy Bridge core is 7.54 W. Therefore, we estimate the area of the PickHead module to be 1.27% of a Sandy Bridge core and 1.09% of a Power7+ core. We estimate the power consumption of the PickHead module to be 0.81% of a Sandy Bridge core.

2.7 Conclusion

Priority-based task scheduling algorithms pose a tradeoff: alleviating the synchronization hotspot by using relaxed PQs leads to wasted work. To

address this tradeoff, we introduce the Snug architecture. Snug distributes the PQ into subqueues, maintains a set of Work registers that point to the highest-priority task in each subqueue, and provides an instruction that picks a high-quality task to execute.

We evaluated Snug on a simulated 64-core chip. We compared the execution time of graph and discrete event simulation applications under Snug and several other PQ algorithms. We found that Snug selects high-quality tasks while avoiding hotspots, minimizes wasted work, and consumes acceptable network bandwidth. Snug reduces the average execution time of the applications by 1.4×, 2.4×, and 3.6× compared to concurrent skip list, SprayList, and software-distributed PQs, respectively. Moreover, compared to the latter two relaxed PQ designs, Snug reduces the number of wasted tasks by 3.3× and 36.8×, respectively.

CHAPTER 3

V-COMBINER: SPEEDING-UP ITERATIVE GRAPH PROCESSING ON A SHARED-MEMORY PLATFORM WITH VERTEX MERGING

3.1 Introduction

Many graph processing applications used in the machine learning, social network, computational biology, and financial system domains are inherently iterative. Examples include belief propagation [50, 51], community detection [52], page rank [53], and hyperlink-induced topic search [54]. They typically have a property that gets propagated across vertices, and a metric for convergence to a solution, which determines their time complexity.

Iterative algorithms are inherently costly for large graphs. For example, running page rank on billion-vertex graphs can take minutes in a 24-core shared-memory system [55]. In many environments, this latency is unacceptable. On the other hand, many applications can trade a small amount of accuracy for improved performance [56, 57]. For example, in page rank, the user is often interested in a very small subset of the computation results [58, 59] — almost all queries only require to know the top-ranked pages. Further, vertex classification algorithms such as community detection and belief propagation can tolerate small errors as long as the final inferred vertex labels remain correct.

Providing high accuracy in approximate iterative graph computations is challenging, as the error introduced in one vertex can propagate to its neighbors (and eventually to all other reachable vertices), and gets accumulated across iterations. For this reason, approximations used in non-iterative graph algorithms such as single-source shortest path [60] or triangle counting [61] are not suitable for iterative graph algorithms.

Previous works for iterative graph algorithms [62, 63, 64, 55] apply program approximation techniques such as loop perforation [65] and task skipping [66] to the input graph [64, 55] or to the program [63, 62]. These techniques speed-

up the program. However, by dropping random vertices or connections in the graph, or skipping their computation as the program runs, these techniques introduce non-determinism, which makes debugging difficult and may cause variability in the outputs of the application.

Graph summarization techniques [67] generate a simpler form of the graph for faster processing and/or more efficient storage. The most popular of these techniques include sketching [68, 69, 70, 71], sparsification [56, 72] and k-core decomposition [73, 74, 75]. Sketching techniques summarize the graph information into a data structure that can be queried in the future to provide a fast approximate answer after a few simple computations. However, sketching techniques have a large overhead and, therefore, cannot be used for performance-intensive tasks. The sparsification and k-core decomposition techniques have a much smaller overhead. However, the resulting graphs have performance and accuracy limitations for certain applications. As we will see, these two techniques are unable to achieve both high speedup and high accuracy.

Our Work. In this chapter, we present *V-Combiner*, a general, deterministic technique for speeding-up iterative graph-processing applications, while maintaining high-accuracy results. V-Combiner is motivated by the observation that not all vertices contribute equally in iterative graph processing algorithms. Therefore, the key technical challenge is identifying and removing the vertices with a small contribution, and still maintaining the graph properties that dictate the accuracy of the graph algorithm.

During pre-processing, V-Combiner *merges* some vertices into their neighboring *hubs*, which are vertices with many connections. It then executes the unmodified original algorithm on the reduced graph, speeding-up execution. Finally, it corrects the final result of the program to account for the effect of the merged vertices.

V-Combiner has the following characteristics:

- **Application Transparency.** Unlike previous work that requires changing the implementation of the graph algorithm [63, 62], in V-Combiner the application is completely unaware of the merging step applied to the input graph. The application treats the approximate graph in the same way as the original graph. This allows the programmer to develop graph algorithms as before.

- **Low Overhead, High Performance, and High Accuracy.** V-Combiner has simpler pre-processing and lower overhead than previous work such as k-core [73, 74, 75], GraphTuner [64] and Input Reduction [55]. V-Combiner speeds up the execution by creating a smaller approximate graph that needs fewer updates. It retains high accuracy by maintaining important graph properties and using a final correction step.
- **Deterministic Behavior.** V-Combiner uses a deterministic merging mechanism that produces identical approximate graphs every time. All previous algorithms except k-core are non-deterministic [64, 55, 62, 63, 56]. Compared to k-core, V-Combiner algorithm maximizes the connectivity of the reduced graph.

Results. We evaluate V-Combiner on a 44-core shared-memory machine running four iterative applications: belief propagation, community detection, hyperlink-induced topic search, and page rank. We execute each application with five real-world data sets. Our main results are:

- Considering the algorithm-time only, V-Combiner speeds-up the computation by an average of 1.55× over the exact baseline algorithm at an accuracy level above 90%, compared to average speedups of 1.46× with sparsification and 1.36× with k-core.
- Considering the end-to-end execution time, which includes one-time overheads, the average speedup of V-Combiner over the exact baseline algorithm is 1.25×, with an accuracy of 91.8%. The performance of V-Combiner is equal to sparsification and significantly better than k-core. Unlike sparsification and k-core, it can successfully meet the accuracy bound on all the benchmarks.
- A trade-off exploration shows that V-Combiner can produce a better set of points in the performance-accuracy trade-off space than the other algorithms in the region above 90% accuracy.
- V-Combiner obtains better load balancing, preserves the average length of the paths, produces deterministic results (unlike sparsification), and performs less work at high accuracy due to high connectivity (unlike k-core).

Contributions. The main contributions of this chapter are:

- We analyze and compare existing techniques for approximating iterative graph algorithms.
- We develop V-Combiner, a novel general approximation technique to improve the performance of iterative graph algorithms with acceptable accuracy losses.
- We evaluate V-Combiner on a shared-memory machine and compare it to state-of-the-art approaches for approximate graph processing.

3.2 Background

A graph G consists of a set of vertices or vertices (V) and edges (E). A graph can be directed or undirected. In directed graphs, an edge can be incoming or outgoing. In undirected graphs, edges do not have a direction. Therefore, each edge can be treated as two logical edges: an incoming and an outgoing one. We refer to the incoming edges of a vertex as *in-edges*, and to outgoing edges as *out-edges*. The vertices at the other end of in-edges are the *in-neighbors* of a vertex. Similarly, the end points of out-edges are the *out-neighbors*. The number of in-edges of a vertex is its *in-degree*, and the number of out-edges is its *out-degree*. For undirected graphs, the degree of a vertex is equal to the number of physical edges of the vertex, and is equal to the vertex's in-degree and to the vertex's out-degree. A *path* in the graph is a sequence of edges that connects a series of distinct vertices.

3.2.1 Iterative Graph Algorithms

Algorithm 3.1 shows the template of an iterative graph algorithm. The algorithm is composed of multiple iterations. In an iteration, the algorithm goes over all of the vertices in the graph (Line 3). For each vertex, it applies an *update function*, which gathers information from the neighbors of the vertex (Lines 5–7) — and potentially from the edges. The value of each neighbor vertex is first multiplied by W, which is a vertex-specific constant. For example, in page rank, W for neighbor vertex v is equal to $\frac{1}{out-degree[v]}$. The result is then aggregated, using an operand that varies across applications (e.g., addition or multiplication) (Line 7). After that, the update function

refines the result stored for the vertex using C_1 and C_2 (Line 8). For page rank, C_1 and C_2 are commonly set to 0.85 and 0.15. In every iteration, a *change* variable (Line 9) accumulates the difference in the values of each vertex before and after the update function. The iterations stop when change is less than a threshold, i.e., the convergence criterion is met (Lines 10–11).

Algorithm 3.1: Iterative graph algorithm.

```
   input  : Graph, Threshold, MaxIter, W, C1, C2
   output: values
1      for i ← 1 to MaxIter do
2          change ← 0;
3          for u in Graph.Vertices do
4              old ← values[u];
5              gatheredVal ← 0 or values[u];
6              for v in Graph.Neighbors(u) do
7                  gatheredVal ← gatheredVal ⊕ (W × values[v]);
8              values[u] ← C1 × gatheredVal + C2;
9              change ← change + | old − values[u] |;
10         if change ¡ threshold then
11             break;
```

The update function creates a notion of information propagation, which can be weights or probabilities. Information propagation generally follows the edges. In undirected graphs, the information propagates in both directions of the edges, whereas in directed graphs, it can propagate in the forward or reverse direction of the edges. Overall, different algorithms differ in the definition of the update function (which is also associated with the direction of edges) and the convergence criterion. In this work, we select four well-known algorithms with different characteristics.

Belief Propagation (BP): BP performs inference on a graphical model [51]. BP calculates a random variable, i.e., the belief, for each vertex, which indicates its most probable state. In BP, the update operation works in both directions, and so propagates the information.

Community Detection (CD): CD finds the community structure of a network [52, 76, 77, 78]. It uses the label propagation method. It takes a directed graph with a set of labeled vertices as input, and propagates the known community IDs to the rest of the vertices in the graph. The update operations happen only in the direction of edges.

Hyperlink-Induced Topic Search (HITS): HITS takes as input a directed graph (e.g., a social or web network) and identifies the most relevant (*aka* authoritative) users/pages given a specific topic [54, 79]. It uses the graph structure to obtain two scores for each vertex, namely the *authority* and *hub*. The authority score of a vertex is computed using the hub scores of its in-neighbors, while its hub score is computed using the authority scores of its out-neighbors. The update operations happen both in the direction of edges to calculate authority scores, and in the reverse direction to compute hub scores.

Page Rank (PR): PR [53] computes the ranks of all the vertices based on the graph structure. The input can be any directed graph, such as a social or web graph. At every iteration, the page rank score of a vertex is computed by summing up the page ranks of its in-neighbors divided by their out-degrees. The update operations happen only in the direction of edges.

Other Graph Algorithms. Our focus is on iterative graph algorithms whose time complexity is determined by their convergence or number of iterations. A number of machine learning algorithms on graphs also belong to this category. However, our observations and techniques do not apply to triangle counting, single-source shortest path, betweenness centrality, and similar problems because their time complexity is independent of the number of iterations. Moreover, it is more challenging to provide high accuracy for iterative algorithms, as the error in one vertex gets propagated and accumulated to all the other reachable vertices across iterations. Hence, approximations proposed for algorithms such as single-source shortest path [60] and triangle counting [61] are not suitable for iterative graph algorithms. Finally, the running times of non-iterative algorithms are relatively low compared to the iterative ones. Hence, there is less opportunity to accelerate them using approximations.

3.2.2 Graph Pre-processing

A graph application has two main parts: pre-processing and computation. Existing work often ignores the pre-processing time and only focuses on optimizing the computation. However, there exists a trade-off between pre-processing time and algorithm execution time [80]. Thus, it is important to

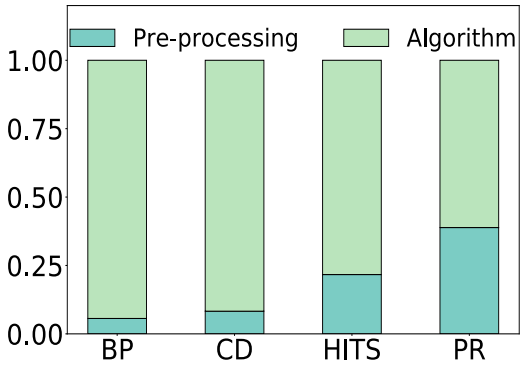


Figure 3.1: Impact of pre-processing time.

account for the pre-processing time when optimizing computation time. Figure 3.1 shows that the pre-processing time is a significant fraction of the total execution time for some of our applications. These results are measured on a machine with 44 threads (Section 3.7.1).

3.3 Observations

V-Combiner is based on several observations we make on graphs.

3.3.1 Not All Vertices Contribute Equally

In Algorithm 3.1, the number of updates generated at a vertex is equal to the number of neighbors it has. For directed graphs, it is equal to the in-degree if updates happen in the direction of edges, or to the out-degree if they happen in the reverse direction. Therefore, vertices with higher degrees have a higher impact on how information is propagated throughout the graph.

Typically, in real-world social and web graphs, a significant portion of the vertices have small degrees. For example, for the large Twitter [81] and PLD [82] graph datasets, we find that, on average, 97.4% of the vertices are connected to at most 100 other vertices. In contrast, 0.00005% of the vertices are connected to at least 100,000 vertices. They are typically identified as *hubs*, and substantially impact how information is propagated.

Figure 3.2 shows an example subgraph. It has three vertices that are highly connected (①,③, and ④), and one that is not (②). The former have more impact on information flow.

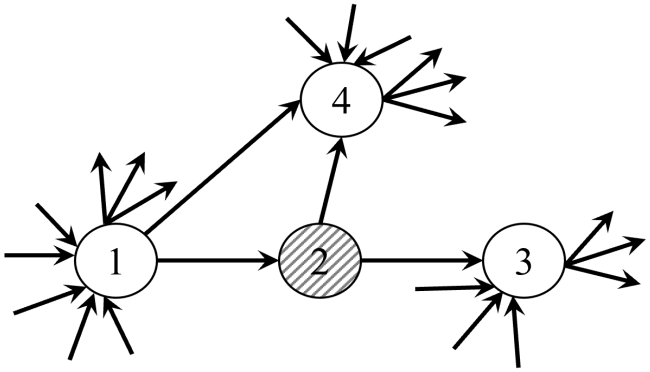


Figure 3.2: Illustration of different vertex connectivity.

3.3.2 A Subset of Results Is All That Is Needed

In many real-life applications of graph processing, the application computes over many vertices, but the user is only interested in a very small subset of the results. The subset of most popular vertices is enough in many scenarios. For example, a company may want to identify its most influential customers, or a researcher may want to find the top authoritative researchers in some domain in the DBLP network [59] or the top authoritative pages in a hyperlink environment [54]. In fact, the page rank algorithm in [58] returns only the top ranked vertices. Therefore, it should often be acceptable for an approximate implementation of the algorithm to return the same top ranked vertices even though the ranking of the less popular vertices may be different.

3.3.3 Applications Can Tolerate Small Errors

In applications such as belief propagation and community detection, the goal is to infer the broad category of each vertex, rather than to compute an exact value for each vertex. Each vertex has a vector of values, where each element of the vector corresponds to the probability of the vertex belonging to a specific category. The highest probability indicates the category of the vertex. As such, small errors in the probability values can be tolerated as long as the inferred vertex category remains the same.

3.3.4 Removing Some Vertices Can Be Effective

Removing some vertices or edges reduces the number and cost of update operations. If the high-level structure of the graph is preserved, the computation on the reduced graph could be a close approximation of that on the original graph. Our intuition is that, since *hub* vertices shape most of the information flow in the graph, vertices from their neighborhood that have a small degree can be *merged* into the hubs, and the overall flow of information in the graph be only marginally affected. Merging a vertex into a hub involves removing the vertex and adding all (or a subset) of its edges to the hub vertex.

3.4 Limitations of Current Techniques

For the types of algorithms that we consider, there are two main existing techniques that prune graphs. They are *sparsification* [56, 72] and *k-core* [73, 74, 75]. These techniques have a few limitations, which act as a motivation for our work. In this section, we discuss these limitations. Later, in Section 3.7.1, we improve the two techniques and, in Section 3.8, evaluate them against V-Combiner.

Sparsification selects some edges in the graph using a probabilisic method and prunes them from the graph. It does not prune any vertices. The pruned edges are chosen based on a sparsification parameter, the shape of the graph, and some properties of the edges. The pruning can be made more or less aggressive. Details are provided in Section 3.7.1.

Since only edges are removed and not vertices, the algorithm still has to loop over all the vertices. In addition, by removing edges, it typically increases the length of the paths between vertices. As a result, the number of iterations that a graph algorithm requires to converge typically increases. For these reasons, it may be difficult to obtain a high speedup with a high algorithm accuracy.

K-core takes a graph and removes as many vertices as needed so that the remaining vertices have a degree equal or greater than k. The result is the k-core graph. This technique prunes both vertices and edges. Higher values of k imply that more vertices and edges are dropped.

Since both edges and vertices are removed, this technique can deliver higher

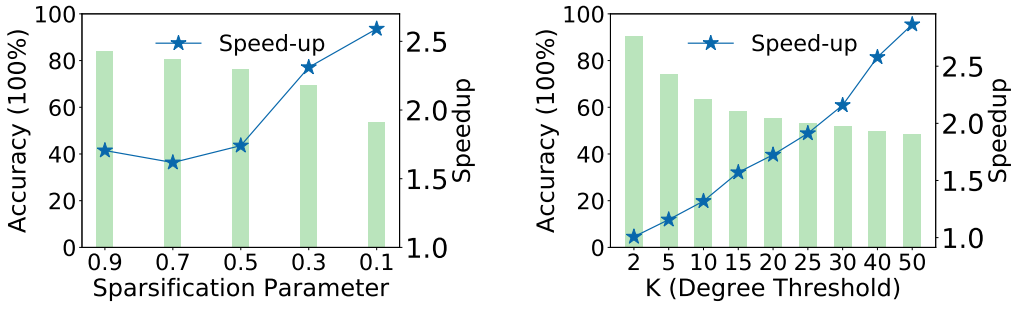
speedups. It reduces the work without increasing the number of iterations until convergence. However, the downside is that, as k increases, more and more of the vertices remaining in the graph become disconnected. As the graph loses connectivity, the accuracy of the algorithm suffers.

As an illustration, Figures 3.3 (a) and 3.3 (b) show representative scenarios. They correspond to runs of page rank (PR) using sparsification (Figure 3.3 (a)) and community detection (CD) using k-core (Figure 3.3 (b)). Section 3.7 describes the input graph used (PLD), the parameters used, and the 44-core machine running the experiments. The figures show, for different levels of pruning (X axis, where the level of pruning increases toward the right side), the accuracy as defined in Section 3.7 (left Y axis) and the speedup relative to the execution using the full graph (right Y axis).

We observe that, as the extent of pruning increases, the speedup increases, but the accuracy decreases substantially. For these applications and input graph, we see that even with a small degree of pruning (left side of X axis), the accuracy is already 90% or lower and, for k-core, the speedup is minuscule.

3.5 Vertex Merging with V-Combiner

Based on the previous discussion, an effective approach to reduce the graph size needs to involve removing both edges and vertices. In addition, as we remove them, we have to be careful to eliminate (or at least reduce to a minimum) the possibility of disconnecting parts of the graph. Consequently,



(a) Sparsification with PR

(b) K-core with CD

Figure 3.3: Illustration of sparsification and k-core.

our proposal is to merge some vertices into nearby hubs, adding some of the edges of the removed vertices to the hub. The result is both accuracy and speedups.

Our technique is called *V-Combiner*. V-Combiner proceeds in two steps. First, it runs a *vertex merging* algorithm during graph pre-processing time, to identify vertices that have few connections and can be merged into their neighboring hubs. Then, after the application completes execution, it runs a correction or *recovery* algorithm to quickly obtain acceptable values for the merged vertices.

Figure 3.4 shows the end-to-end execution timeline for both the conventional exact approach and V-Combiner. The exact approach includes pre-processing time (i.e., when the graph is built) and compute time (i.e., when the graph algorithm runs). The V-Combiner approach contains pre-processing time (which includes vertex merging and building the graph), compute time (i.e., when the graph algorithm runs), and post-processing time (i.e., when the recovery takes place). The shaded boxes are V-Combiner-specific steps.

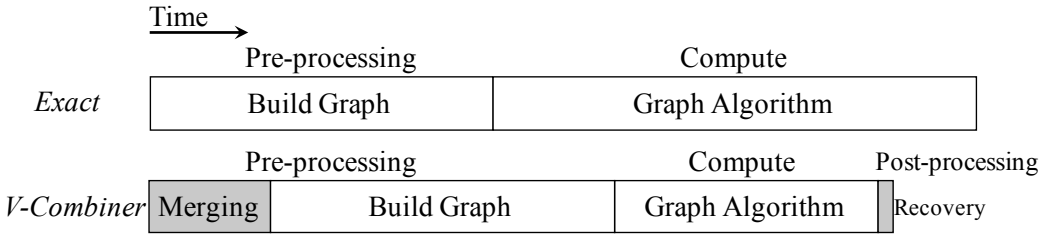


Figure 3.4: End-to-end execution steps of a graph algorithm.

The performance benefits of V-Combiner come from the reduced algorithm time due to the reduced number of vertices and edges. More specifically, the graph algorithm can converge with fewer iterations (because of the reduced vertices) and fewer update operations in each iteration (because of the reduced edges). Ideally, the reduction in algorithm time should more than offset the additional overheads of the pre- and post-processing steps.

Support for Different Graphs and Algorithms. V-Combiner can be applied to both directed and undirected graphs with minimal changes. Moreover, it can be used for a variety of graph algorithms with different types of update propagation. For example, in page rank, updates are always propagated from in-neighbors, whereas in HITS [54], updates are propagated both

from in- and out-neighbors. Based on the update propagation, we classify algorithms into *one-way* propagation and *two-way* propagation. For undirected graphs, the propagation is naturally two-way. When merging, V-Combiner takes into account the direction of update propagation used by the algorithm, and tries to maintain the connectivity of the graph. The goal is for the updates to continue to propagate through the edges even after certain vertices are removed.

3.5.1 Merging Approach

During the merging step, V-Combiner proceeds to identify a subset of the high-degree vertices that we call *supernodes*. Then, for each supernode, it considers the vertices in its input neighborhood (or in its output neighborhood, if the updates are propagated from out-neighbors). If these vertices have low in- and out-degrees, we say that they are *subnodes* of the supernode. The assumption is that the contribution of such vertices to the final result of the algorithm is likely to be small. Hence, the subnodes are merged into the supernode, effectively being pruned away. The supernode takes some of the edges of the merged subnodes, as explained in Section 3.5.2.

The resulting graph has a set of supernodes with now higher degrees, connected with what we call *Regular* vertices. The resulting graph has a similar structure to the original one, but with many fewer vertices and edges.

We now give the precise definitions of supernodes, subnodes, and regular vertices in a given graph G. In the definitions, we use three thresholds called α, β, and π. The α and β thresholds are the lower and upper thresholds, respectively, for the degree of a vertex that qualifies as a supernode. The π threshold is the upper threshold for the degree of a vertex that qualifies as a subnode. We describe how to obtain such thresholds in Section 3.5.6.

Definition 1: Supernodes (V_{sup}) are vertices with an in-degree higher than α and lower than β,

$$V_{sup} = \{\ v \mid v \in G.V\ \wedge\ \mathit{InDegree}(v) > \alpha\ \wedge\ \mathit{InDegree}(v) < \beta\ \}.$$

Vertices with in-degree higher than or equal to β are not considered supernodes. This is because we do not want them to increase their degree further by taking-in edges from merged subnodes. Vertices with very high degree can cause load imbalance and slow down the execution. Hence, these vertices remain unchanged in the graph.

For two-way algorithms in directed graphs, supernodes are vertices where the sum of in-degree and out-degree is higher than α and lower than β. Further, a supernode is an *in-supernode* if its in-degree is higher than or equal to its out-degree, and an *out-supernode* if the opposite is true.

For undirected graphs, supernodes are vertices with a degree between α and β.

Definition 2: Subnodes (V_{sub}) are vertices that have an in-degree lower than π (where π ¡ α), an out-degree lower than π, and at least one output that connects them to a supernode,

$$V_{sub} = \{\ v \mid v \in G.V \wedge \mathit{InDegree}\,(v) < \pi \wedge \mathit{OutDegree}\,(v) < \pi \\ \wedge \exists w \in V_{sup}\ .\ w \in \mathit{OutNeigh}\,(v)\ \}.$$

A vertex with an in-degree or an out-degree higher than π is not a subnode because it is too important to be merged into a nearby supernode. Also, it is possible that a subnode is connected to two supernodes; in this case, it can be merged into either with a deterministic algorithm.

For two-way algorithms in directed graphs, subnodes are vertices where the sum of in-degree and out-degree is less than π, and have at least one edge that connects them to a supernode. Further, a subnode is an *in-subnode* if it has at least one output that connects it to an in-supernode, and is an *out-subnode* if it has at least one input that connects it to an out-supernode. A subnode can be both an in-subnode and an out-subnode.

For undirected graphs, subnodes are vertices with a degree lower than π and at least one edge that connects them to a supernode.

Definition 3: Regular vertices (V_{reg}) are the remaining vertices,

$$V_{reg} = G.V - (V_{sub}\ \cup\ V_{sup}).$$

V-Combiner leaves those vertices intact.

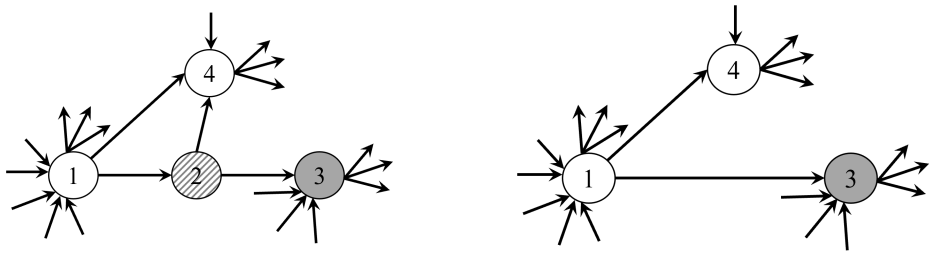
3.5.2 Vertex Merging Algorithm

V-Combiner performs vertex merging to generate the approximate graph. The algorithm involves merging each subnode into a neighboring supernode.

In the following, without loss of generality, we describe the vertex merging algorithm assuming one-way graph algorithms in directed graphs where updates are propagated from in-neighbors. In Section 3.5.5, we describe scenarios with other types of directed and undirected graphs.

Merging a subnode into a supernode involves removing the subnode and altering the edges as follows. First, the input edges of the subnode now become input edges of the supernode. This operation may create duplicated edges, i.e., multiple edges connecting the same input vertex to the same output vertex; such duplication will be later eliminated when the graph is built. Second, the output edges of the subnode are dropped. V-Combiner chooses this design over a possible alternative where the output edges of the subnode become output edges of the supernode. Such alternative is undesirable because it could create a new, previously-nonexistent path.

Figure 3.5 shows an example of vertex merging. Figure 3.5 (a) shows an original subgraph, where V-Combiner wants to merge subnode ② into supernode ③. Figure 3.5 (b) shows the approximate graph after merging.



(a) Original (b) Approximate

Figure 3.5: Vertex merging in V-Combiner.

As V-Combiner merges subnode ② into ③, it transforms edge ① → ② into ① → ③. Next, V-Combiner discards edge ② → ④. Had V-Combiner chosen to transform it into an edge ③ → ④, it would be creating a new, previously nonexisting path between ③ and ④. Hence, V-Combiner does not create such an edge.

Algorithm Description. Algorithm 3.2 shows the pseudo code of the merging algorithm in V-Combiner. The algorithm takes as input the number of vertices, the list of edges, and the in-degrees and out-degrees of the vertices. The output of the algorithm is the new in- and out-degrees of the vertices, and an array called *superNodes*. This array has as many entries as vertices

in the original graph. If a vertex has become a subnode, its entry in *superNodes* has the ID of its supernode; if a vertex has become a supernode, its entry in *superNodes* has its own ID; for the remaining vertices, the entry in *superNodes* holds -1. The output of the algorithm will be later used in the step that builds the graph.

Algorithm 3.2: Merging algorithm in V-Combiner.

inputs : N (number of vertices), edges (list of edges), inDegrees, outDegrees, α, β, π
outputs: newInDegrees, newOutDegrees, superNodes

```
1   for i ← 0 to N − 1 do
2       if inDegrees [i] > α and inDegrees [i] < β then
3           superNodes [i] = i
4   for e in edges do
5       if superNodes [e.dst] = e.dst
6           and inDegrees [e.src] < π
7           and outDegrees [e.src] < π then
8           superNodes [e.src] = e.dst;
9           newInDegrees [e.src] ← 0;
10          newOutDegrees [e.src] ← 0
11  for e in edges do
12      if e.dst is a subnode and e.src is NOT a subnode then
13          newInDegrees [superNodes[e.dst]] += 1
14      if e.src is a subnode and e.dst is NOT a subnode then
15          newInDegrees [e.dst] -= 1
```

The merging algorithm consists of three sections. Each section can be executed in parallel:

- *Identify supernodes:* First, all vertices are scanned in parallel to identify any vertex that has an in-degree higher than α and lower than β (Lines 1–3).
- *Identify subnodes:* The second parallel section (Lines 4–10) consists of processing the list of edges in parallel. For every edge e, such that $e.dst$ is a supernode, we check if $e.src$ qualifies to become a subnode. If so, the *superNodes* entry of $e.src$ is set to $e.dst$. Also, we prune the in- and out-edges of $e.src$. Therefore, the new in- and out-degrees of $e.src$ are set to 0.
- *Merge Vertices and Update Degrees:* Finally in Lines 11–15, the algorithm

computes the new in- and out-degrees of all affected vertices. Recall that merging affects both the in-neighbors and the out-neighbors of the subnode. Each in-neighbor has to be connected to the supernode. Hence, we increase the in-degree of the supernode in Lines 12–13. Moreover, the out-edges of the subnode have to be eliminated, and the out-neighbors have to update their in-degrees (Lines 14–15).

Delta Graph Construction. After merging, when the graph is built, V-Combiner also constructs a small graph named the *delta* graph. The delta graph contains only the subnodes and their immediate in-neighbors. The delta graph will be used to generate good final values for the subnodes in the recovery step.

3.5.3 Recovery Step

Recall that the subnodes do not have any values after the graph algorithm completes. The goal of V-Combiner's recovery step is to assign the final values to these subnodes.

To compute the values of the subnodes, the recovery algorithm takes the delta graph and proceeds as follows. The in-neighbors of subnodes in the delta graph are given the values computed by the computation on the approximate graph. Such values are close to their exact values. Then, we run the recovery algorithm, which is specified by the developer. It simply uses the basic operator of the corresponding graph algorithm to generate the values of the subnodes using the values of their in-neighbors in the delta graph.

3.5.4 Overall V-Combiner Algorithm

Algorithm 3.3 shows the overall V-Combiner algorithm. First, the subnode vertices are merged using Algorithm 3.2. Then, the output of the Merging algorithm, including the *superNodes* array, is passed to a build step. This step constructs both the delta and the approximate graphs (Line 2). After that, the graph algorithm, outlined in Algorithm 3.1 executes on the approximate graph using the user-specified algorithm parameters (Line 3), and returns its results. Finally, in Line 4, the recovery algorithm receives these results and recovers the values for the subnode vertices using the delta graph.

Algorithm 3.3: Overall V-Combiner algorithm.

```
input : merging_arguments, algo_params
output: final_results
1 merging_output = Merging(merging_arguments);
2 approx_graph, delta_graph = Build(merging_output);
3 results = GraphAlgo(approx_graph, algo_params);
4 final_results = Recovery(results, delta_graph, algo_params);
```

3.5.5 Other Scenarios of the Merging Algorithm

Section 3.5.2 only described how to perform merging for the most common scenario, i.e. directed graphs and one-way algorithms. In the example algorithm, V-Combiner picks subnodes from the inputs of the supernode, and merges subnodes into supernodes in the forward direction of the edges. In the case where information flows only in the reverse direction of the edges in a directed graph, V-Combiner would pick subnodes from the outputs of the supernode. However, we are unaware of an example for this scenario.

Table 3.1 shows a taxonomy with two other possible scenarios that V-Combiner can support: (i) directed edges where the information flows in both directions; (ii) undirected edges. V-Combiner supports both scenarios using the same merging techniques.

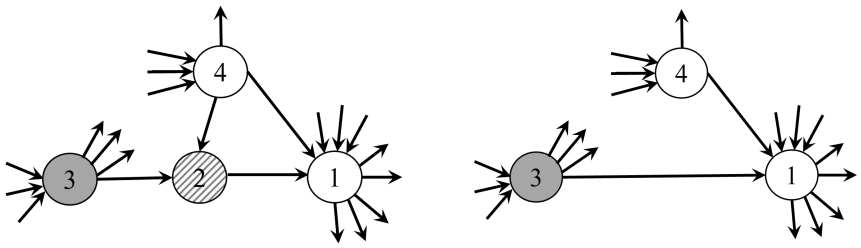
Table 3.1: Graph processing scenarios V-Combiner supports.

Example application	Edges	Information flow
page rank, community detection	Directed	One-way
hyperlink-induced topic search	Directed	Two-way
belief propagation	Undirected	Two-way

Directed graphs and two-way algorithms. Since the goal of V-Combiner is to preserve connectivity in the direction of the update propagation, in this case where updates propagate in both directions, V-Combiner merges subnodes into supernodes in both directions. Recall from Section 3.5.1 that supernodes are vertices where the sum of in-degree and out-degree is higher than α and lower than β, and that subnodes are vertices where the sum of in-degree and out-degree is less than π, and have at least one edge that connects them to a supernode. We further classified supernodes into in-supernodes

and out-supernodes, and subnodes into in-subnodes and out-subnodes. Intuitively, in-supernodes are better connected through their inputs (compared to their outputs), and out-supernodes are better connected through their outputs (compared to their inputs). In these graphs and algorithms, V-Combiner merges in-subnodes into in-supernodes, and out-subnodes into out-supernodes. We call the first kind of merging *forward merging* and the second kind *reverse merging*.

Figure 3.6 shows an example of reverse merging, where out-subnode (2) is merged into out-supernode (3). In this case, edge (2) → (1) is transformed into (3) → (1), and edge (4) → (2) is dropped.



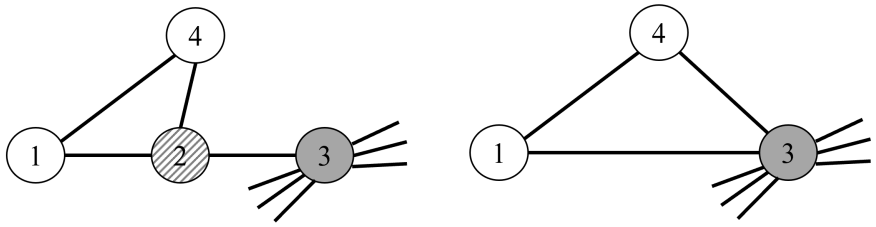
(a) Before merging (b) After merging

Figure 3.6: Merging out-subnode (2) into out-supernode (3).

Intuitively, merging in both directions provides better connectivity than merging only in one direction. We perform forward merging of subnodes into supernodes that have better connectivity through their inputs, and reverse merging of subnodes into supernodes that have better connectivity through their outputs.

Undirected graphs. Recall from Section 3.5.1 that supernodes are vertices whose degree is higher than α and lower than β, and that subnodes are vertices whose degree is less than π, and have at least one edge that connects them to a supernode. In this algorithm, when a subnode is merged into a supernode, all the edges of the subnode are added to the supernode. V-Combiner does not drop any edges because paths do not have a specific direction. Duplicate edges are removed. This operation is also known as vertex-contraction [83].

Figure 3.7 shows how subnode (2) is merged into supernode (3) and its edges are also added to (3). All the edges of the former are added to the latter.



(a) Before merging (b) After merging

Figure 3.7: Merging subnode (2) into supernode (3).

3.5.6 Choosing the Merging Parameters

To effectively reduce a graph, V-Combiner needs to choose a good set of supernodes. The number of supernodes is directly controlled by thresholds α and β. To find good values of α and β, we perform the following experiment. We first rank the vertices based on their in-degree, from lower to higher. We then accumulate the number of edges of the vertices, in order.

Figure 3.8 shows the resulting cumulative distribution function (CDF) of edges as a function of the in-degrees of vertices in the Friendster graph [84]. We divide the plot into three regions. In the leftmost region, the curve has a sharp slope. This part accounts for the majority of the edges. These edges connect many vertices with small in-degrees.

In the second region, the curve goes through the knee. Finally, in the third region, the curve flattens up. We argue that supernodes should be chosen from the knee region. This region has many vertices with substantial, but not excessive, in-degrees. Supernodes should not be chosen from the third region. In this region, vertices have very large in-degrees. Increasing their in-degrees further is likely to cause load imbalance.

In Section 3.8.5, we study different ranges for the knee region, to pick the most profitable one. The boundaries of such a region are the values of α and β.

Once we have picked the values of α and β, we need to pick a value of π. Higher values of π mean that more vertices qualify as subnodes. More subnodes typically translates into lower accuracy, because the elimination of subnodes with large in- and out-degree affects many neighboring vertices. However, more subnodes also translates into more time savings in processing

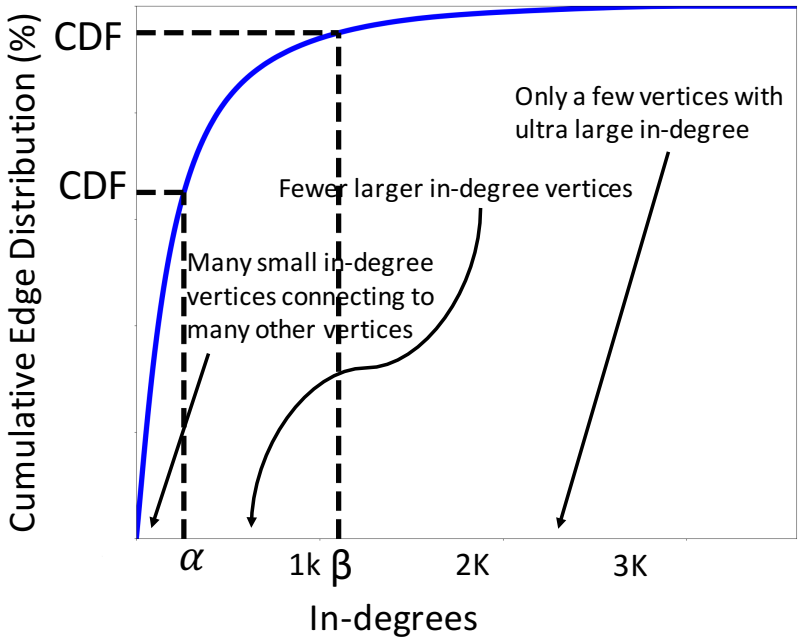


Figure 3.8: Finding α and β based on the cumulative edge distribution.

the graph. Therefore, given α and β values, π can be considered as a knob to directly control the trade-off between accuracy and speedup.

3.6 Comparison of Techniques

Table 3.2 qualitatively compares sparsification, k-core, and V-Combiner in terms of their impact on connectivity, length of paths, load balancing, pre- and post-processing overhead, and overall speedup.

Table 3.2: Qualitative comparison of different techniques. In the table, a check mark means that the technique is doing well; a cross means the opposite.

Technique	Connectivity	Length of Paths	Load Balancing	Pre/Postproc. Overhead	Overall Speedup
Sparsification	✔	✘	✘	✔✔	✔
K-core	✘	✘	✔	✘	✘
V-Combiner	✔	✔	✔	✔	✔

Connectivity. To attain high accuracy, the reduced graph should strive to maintain connectivity between the remaining vertices. As discussed in Section 3.4, aggressive k-core easily ends-up causing graph disconnectivity. Sparsification and V-Combiner do not. V-Combiner minimizes disconnectiv-

ity by merging a subnode into a hub, and making the in-neighbors of the subnode to become in-neighbors of the hub.

Average Length of Paths. Retaining the length of the paths between vertices in the reduced graph is sometimes important for performance and accuracy. Specifically, if paths are longer, graph algorithms typically take more iterations to converge. Furthermore, in some algorithms, changing the length of the paths causes distortions that result in low accuracy. Generally, all these three algorithms change the average length of the paths. As discussed in Section 3.4, since sparsification removes edges, it typically increases the length of the paths between vertices. Since k-core and V-Combiner remove vertices, they tend to reduce the average length of the paths. We observe, however, that V-Combiner is the one that changes the average length of the paths the least. Intuitively, this is because it includes two opposite effects: it reduces the length of paths by removing some vertices, but increases the length of paths by dropping some edges.

Load Balancing. To attain high speedups, graph algorithms should keep a good balance of the load between threads. As shown in Algorithm 3.1, a graph algorithm can be parallelized by assigning a subset of the vertices to each thread to process. The amount of work performed per vertex is proportional to its degree. It is well known that many graphs present a very skewed distribution of the vertices: there are many small-degree vertices and few high-degree ones. We observe that both k-core and V-Combiner improve load balancing by making the distribution less skewed. Specifically, they remove many of the small-degree vertices, either by pruning (in k-core) or by merging (in V-Combiner). Sparsification largely keeps the same distribution.

Pre- and Post-Processing Overhead. To attain good speedups, it is important that the graph reduction techniques have minimal impact on the pre- and post-processing steps, including the building of the graph. Sparsification has the least pre-processing overhead. k-core has the largest pre-processing overhead: even though we use the state-of-the-art k-core implementation [85], pruning until the remaining vertices have a degree of at least k has substantial overhead. The overhead of the pre- and post-processing in V-Combiner is higher than in sparsification. We show the numbers in Section 3.8.

Overall Speedup. Based on all the factors considered, the last column of Table 3.2 outlines the expected relative performance of the techniques.

3.7 Experimental Setup

We implement V-Combiner in C++ using OpenMP. We use the page rank code from GAP [4], and implement community detection [76, 52], belief propagation [51], and hyperlink-induced topic search [54] ourselves.

3.7.1 Methodology

Machine Specification and Graph Datasets. To evaluate the effectiveness of V-Combiner and the other techniques, we perform experiments on a two-socket shared-memory system with 44 Intel Xeon Gold 6152 cores and 192 GB of memory. We disable the dynamic voltage and frequency scaling (DVFS) mechanism to minimize run-time variation, and use the *numactl* library with the *interleave all* option to average out the numa effects on the programs. Table 3.3 shows our graph datasets, which are chosen from several different domains. The belief propagation (BP) benchmark uses undirected graphs. Unfortunately, we could not run BP on our largest two graphs, namely FS and TW, as their memory requirements exceed the machine's memory capacity.

Table 3.3: Graph datasets.

Graph dataset	Vertices	Directed Edges	Undirected Edges
Friendster (FS) [84]	65.6M	1715.7M	-
Twitter (TW) [81]	61.5M	1456.1M	-
Page-Level Domain (PLD) [82]	42.9M	623.1M	582.6M
Arabic-2005 (AR) [86]	22.7M	631.2M	553.9M
Dbpedia (DB) [84]	18.3M	136.5M	126.9M

Evaluation Configurations. We perform end-to-end execution time analysis of our benchmarks. We run each benchmark using both exact and approximate graphs in several different configurations. For each configuration, we measure the time spent in each step of execution by averaging out the results of five runs. The following steps are measured:

- **Prune/Merge.** Identify the vertices/edges to prune or merge.
- **Build.** Construct the approximate (and delta) graph.
- **Algorithm.** Execute the graph algorithm.

- **Recovery.** Generate the values of the eliminated vertices (V-Combiner and k-core only).

  We evaluate the following configurations:

- **Exact.** This is the baseline execution, which is running on the unmodified (original) graph dataset. All the approximate executions are compared to this baseline. The output of this execution is also used as the ground truth for evaluating accuracy.

- **Sparsification.** Sparsification prunes one of many edges from vertices with large degrees. For each edge (u, v), it calculates the probability of keeping it using the formula $\frac{S \times d_{avg}}{min(d_o^u, d_i^v)}$. Here, S is the sparsification parameter, which is suggested to be in the interval [0.1, 0.9] [56]. Also, d_{avg} is the average degree of the graph, defined as the ratio of the number of edges over the number of vertices. d_o^u and d_i^v are the out-degree and in-degree of the vertices at the two ends of the edge, respectively. The denominator is the minimum of the two degrees. The equation applies to undirected graphs too. Unlike V-Combiner, sparsification does not require a recovery step, since the values for all the vertices are computed on the approximate graph.

  Optimization: Figure 3.3 (a) shows that the accuracy achieved by sparsification at 0.9 sparsification is 83%. This low accuracy is because a significant number of edges were dropped. To fix this problem, in our evaluation in Section 3.8, we constrain sparsification using a second parameter (*Percent_E_Remain*), which enforces that a certain fraction of edges remain in the approximate graph.

  Sweeping parameters: We use three sparsification parameter S values (0.9, 0.7, and 0.5), each with five different *Percent_E_Remain* parameter values (90%, 80%, 70%, 60% and 50%).

- **K-core.** We follow the state-of-the-art implementation of k-core [85] to identify all the vertices that have to be dropped given a certain k. When a vertex is dropped, all of its in- and out-edges are dropped too.

  Optimization: In our evaluation in Section 3.8, we add a recovery step to assign values to the removed vertices. This is similar to the algorithm described in Section 3.5.3 for V-Combiner.

  Sweeping parameters: We try k values equal to 2, 5, 10, 15, 20, 25, 30, 40,

and 50. We only consider k-core configurations that have a percentage of remaining edges over 50%.

- **V-Combiner.** V-Combiner uses a combination of pruning and merging, unlike sparsification and k-core, which are pruning-only.

  Sweeping parameters: The (α, β) interval is selected to create a subrange of edge CDF in the curve of Figure 3.8 within the (65%, 95%) interval. This corresponds to the knee region of the curve. For all benchmarks except HITS, we consider intervals of length 10% each, i.e. (65%, 75%), (75%, 85%) and (85%, 95%). HITS is a two-way algorithm with directed edges and, therefore, there is more merging and edge dropping. For this reason, we experiment with half-sized intervals for HITs, i.e. (65%, 70%),... , (90%, 95%). Given an interval, we sweep the value of π so that the CDF of edges ranges from 5% to CDF(α)%-5%. Finally, we only consider configurations that have a percentage of remaining edges over 50%.

3.7.2 Accuracy Metrics

We target an accuracy threshold of 90% for all the benchmarks, which is a common threshold used by past work [87, 65].

CD. In CD, each vertex will be assigned a label, indicating the probability of that vertex to belong to a specific community. To measure accuracy, we compute the fraction of initially-unlabeled vertices that end-up being identified to belong to the correct community. We determined the correct community by the exact computation on the original graph.

HITS and PR. We measure the *top-k accuracy* [57], which is the fraction of the vertices in the top k ranks of the exact output that are also in the top k ranks of the approximate output. k is application-dependent, and is given relative to the number of graph's vertices.

BP. Based on past work [50, 88, 89, 90, 91], we divide BP use cases into two main categories: (i) classification of vertices based on the class they belong to, and (ii) ranking of the vertices based on information such as user trustworthiness and influence. For the classification scenario, since we are only able to run the algorithm for belief vectors of size 2 (due to not enough memory), we observe high accuracy in most cases. Therefore, we choose the ranking scenario, where we can capture the accuracy better. We use the top-k accuracy metric.

3.8 Evaluation

We first compare V-Combiner's performance-accuracy trade-offs to sparsification's and k-core's, both for algorithm-time (Section 3.8.1) and for end-to-end time (Section 3.8.2). We then present statistics on the approximate graphs (Sections 3.8.3 and 3.8.4), and an analysis of the best pruning or merging parameters (Section 3.8.5).

To compare V-Combiner, sparsification, and k-core, we pick their best parameter configurations from Section 3.7.1 that, for each individual benchmark and input graph, deliver the highest end-to-end speedup over the baseline. The accuracy is required to be equal to or higher than the 90% threshold. We call these configurations the *best end-to-end* configurations.

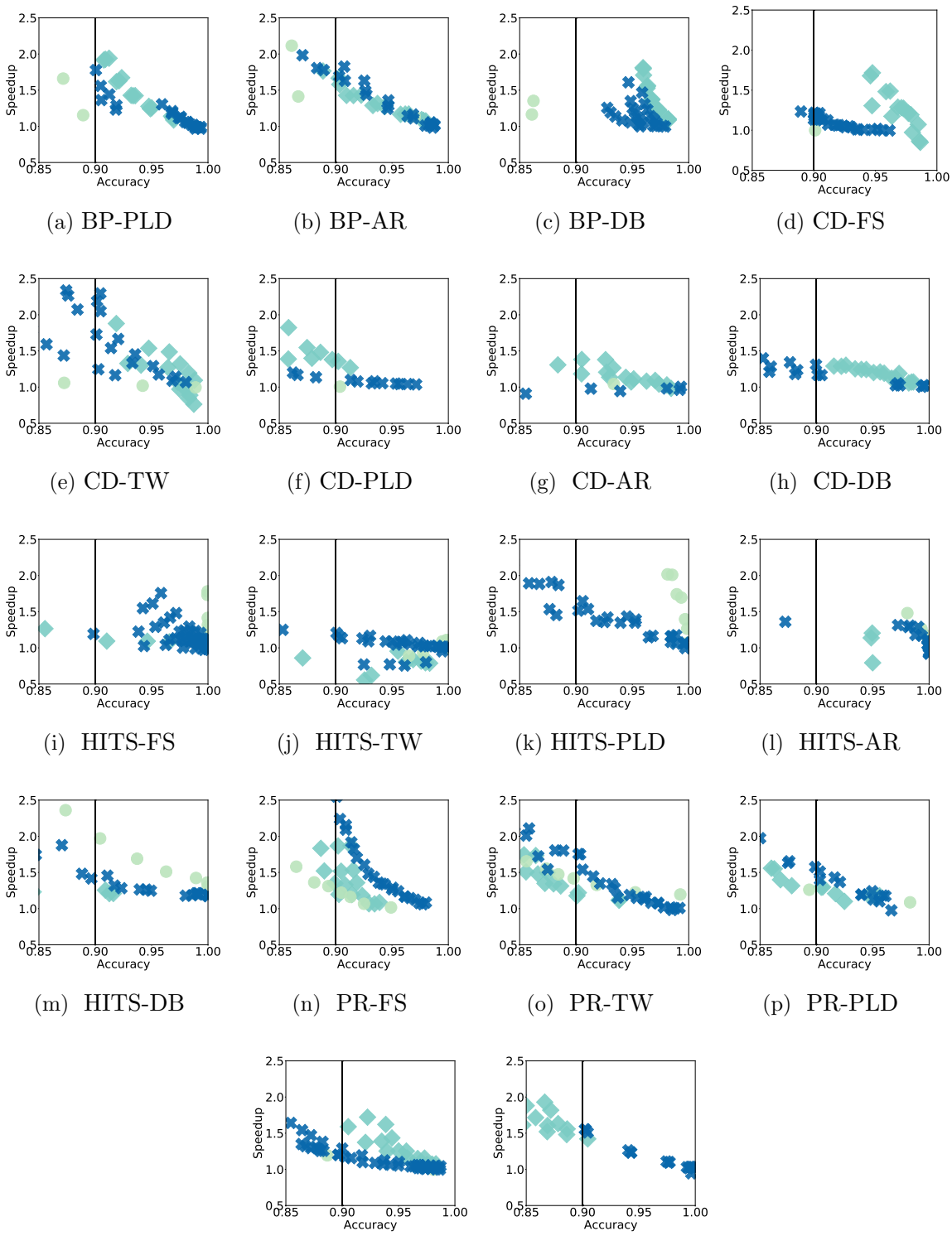
3.8.1 Algorithm Performance and Accuracy

Best Design Points. In this section, to understand the trade-offs between the different algorithms, we take the best end-to-end configurations but recompute the speedups without considering the *pre-processing time*. The "pruning" or "merging" part of the pre-processing time is highly variable across different techniques and overshadows the savings obtained by reducing algorithm time. However, we do consider the post-processing time since it contributes to the final accuracy in both V-Combiner and k-core. In Section 3.8.2, we will analyze the speedups of the best end-to-end configurations with all the times included.

Table 3.4 shows the resulting algorithm execution time (*time*), and the speedup (*sp*), accuracy (*acc*), and percentage of work done (*work*) relative to the baseline for the three techniques. The amount of work done is proportional to the number of edges multiplied by the number of iterations. We also report the "expected" speedup ($\overline{sp}$) next to each speedup number, which is computed as $\frac{100}{work(\%)}$. It largely indicates the speedup that would be achieved without considering any load imbalance effects in the approximate graph.

In the table, each row corresponds to one benchmark and graph. In each row, the technique with the highest speedup is in bold. An empty row means that the technique could not run the benchmark and graph with an accuracy equal to or higher than the threshold.

Comparing V-Combiner to sparsification. V-Combiner attains an average



(a) BP-PLD (b) BP-AR (c) BP-DB (d) CD-FS

(e) CD-TW (f) CD-PLD (g) CD-AR (h) CD-DB

(i) HITS-FS (j) HITS-TW (k) HITS-PLD (l) HITS-AR

(m) HITS-DB (n) PR-FS (o) PR-TW (p) PR-PLD

(q) PR-AR (r) PR-DB

Figure 3.9: Pairs of (speedup, accuracy) for V-Combiner (✖), k-core (●), and sparsification (⧫). Speedups only include algorithm execution time. The vertical line shows the 0.9 accuracy threshold.

Table 3.4: Algorithm execution time (*time*), and speedup (*sp*), accuracy (*acc*), and percentage of work done (*work*) relative to the baseline for the three techniques. Each row corresponds to one benchmark and graph. In each row, the technique with the highest speedup is shown in bold. An empty row means that the technique could not run the benchmark and graph with an accuracy equal to or higher than the threshold.

Benchmark	V-Combiner				sparsification				k-core			
& Graph	time	sp($\overline{sp}$)	acc	work	time	sp($\overline{sp}$)	acc	work	time	sp($\overline{sp}$)	acc	work
BP-PLD	150.51	1.78(1.87)×	90.1%	53.4%	138.88	**1.93**(1.93)×	90.9%	51.7%	-	-	-	-
BP-AR	93.24	**1.83**(2.06)×	90.8%	48.6%	102.20	1.67(1.89)×	90.3%	53.0%	149.90	1.14(1.13)×	97.0%	88.5%
BP-DB	26.05	1.35(1.63)×	94.9%	61.4%	19.39	**1.81**(1.81)×	95.9%	55.2%	-	-	-	-
CD-FS	74.66	1.21(1.24)×	90.0%	80.5%	52.76	**1.72**(2.08)×	94.8%	48.0%	90.59	1.0(1.0)×	90.1%	99.4%
CD-TW	96.29	**2.3**(2.37)×	90.5%	42.1%	117.63	1.88(2.89)×	91.9%	34.6%	216.92	1.02(1.06)×	98.8%	94.2%
CD-PLD	101.49	1.10(1.09)×	91.3%	91.9%	82.27	**1.38**(1.46)×	90.2%	68.5%	110.77	1.0(1.0)×	90.4%	99.8%
CD-AR	64.82	1.0(1.0)×	99.3%	99.9%	47.16	**1.38**(1.82)×	92.6%	54.9%	62.40	1.04(1.07)×	93.3%	93.7%
CD-DB	31.11	**1.31**(1.34)×	90.0%	74.6%	31.26	1.30(1.72)×	92.6%	58.2%	-	-	-	-
HITS-FS	47.42	1.76(1.72)×	95.8%	58.2%	76.29	1.09(1.13)×	94.6%	88.6%	46.79	**1.78**(1.67)×	99.9%	60.0%
HITS-TW	51.67	**1.21**(1.18)×	90.1%	84.8%	65.34	0.95(1.0)×	95.5%	99.6%	56.13	1.11(1.11)×	99.8%	90.0%
HITS-PLD	11.88	1.65(1.54)×	90.6%	64.9%	-	-	-	-	6.93	**2.82**(2.39)×	97.3%	41.8%
HITS-AR	8.42	1.31(1.31)×	97.2%	76.4%	9.72	1.14(1.2)×	94.9%	83.1%	7.48	**1.48**(1.55)×	98.1%	64.4%
HITS-DB	6.78	1.46(1.25)×	91.1%	80.0%	8.17	1.21(1.08)×	91.2%	92.3%	5.01	**1.97**(1.67)×	90.4%	59.8%
PR-FS	15.80	**2.55**(2.62)×	90.1%	38.2%	21.59	1.87(2.03)×	90.2%	49.2%	33.01	1.22(1.22)×	90.5%	81.8%
PR-TW	20.57	**1.75**(1.74)×	90.4%	57.6%	29.53	1.22(1.26)×	90.2%	79.1%	27.14	1.32(1.35)×	91.8%	73.8%
PR-PLD	8.87	**1.51**(1.35)×	90.3%	73.8%	10.36	1.30(1.43)×	90.4%	70.0%	12.35	1.09(1.09)×	98.3%	91.5%
PR-AR	3.96	1.29(1.27)×	90.0%	78.9%	3.21	**1.59**(2.02)×	90.5%	49.4%	5.12	1.0(1.0)×	98.1%	99.6%
PR-DB	0.92	**1.55**(1.89)×	90.3%	52.8%	1.0	1.42(1.45)×	90.5%	69.2%	-	-	-	-
Average	45.25	**1.55**(1.48)×	91.8%	67.6%	48.04	1.46(1.54)×	92.2%	64.9%	59.32	1.36(1.23)×	95.3%	81.3%

speedup of 1.55×, while sparsification attains 1.46×. Surprisingly, this happens regardless of the fact that sparsification performs less work on average, i.e., 64.9% compared to 67.6%. The reason is the better load balancing of the work in V-Combiner due to the merging of small vertices. We observe that, on average in V-Combiner, the speedup is 4.7% higher than the expected speedup, while it is 5.2% lower than the expected speedup in sparsification. Additionally, V-Combiner satisfies the accuracy threshold across all the benchmarks, while sparsification fails to meet the accuracy threshold in HITS-PLD.

V-Combiner ensures high speedups by dropping both vertices and edges, as compared to sparsification which only drops edges. We can observe the higher speedups in all of PR and HITS experiments (except PR-AR) and some of BP and CD experiments, i.e., BP-AR, CD-TW, and CD-DB.

Comparing V-Combiner to k-core. k-core obtains a significantly lower average speedup than V-Combiner, i.e. 1.36×. This is because k-core performs more work than V-Combiner (81.3%) to reach an accuracy of above the 90% threshold. Furthermore, k-core fails to satisfy the accuracy threshold in four experiments. Another interesting observation is that both V-Combiner and k-core achieve average higher speedups than their expected speedups. This

is because both techniques perform small-degree vertex pruning/merging, which helps load balancing the work better than the original graph.

Compared to k-core, V-Combiner relies on the high connectivity of its approximate graph and recovery algorithm to achieve higher speedup-accuracy operating points. The recovery algorithm is more effective in V-Combiner than in k-core. k-core's recovery is performed using other removed vertices with their initial values, while V-Combiner's recovery is performed using vertices that have been already involved in the computation. Except in HITS-FS, HITS-PLD, HITS-AR, and HITS-DB, V-Combiner achieves comparable or higher speedups than k-core. For these graphs, k-core performs less work with higher accuracy.

Overall, V-Combiner attains the highest average speedup in eight out of 18 benchmarks. The reasons are better load balancing and the preservation of the average length of paths (relative to sparsification), and performing less work at high accuracy due to better connectivity (relative to k-core).

Speedup-Accuracy Trends. To further compare V-Combiner, sparsification, and k-core, Figure 3.9 shows pairs of (speedup, accuracy) for the techniques. In the charts, the X-axis shows accuracy from 0.85 (low) and 1.00 (high), and the Y-axis shows speedup over the baseline algorithm. We obtain these points by sweeping the parameters of each technique according to Section 3.7.1. As before, we include the post-processing time but not the pre-processing time, since we focus on the algorithm speedup only. Note that some of the data points in Figure 3.9 show higher speedups than in Table 3.4. This is because Table 3.4 only shows the best *end-to-end* configurations.

Comparing V-Combiner to sparsification. In the figure, the closer the (speedup, accuracy) pairs are to the top right of each plot, the better the trade-off is. For all HITS and PR experiments except PR-AR, V-Combiner achieves better results than sparsification. Typically in PR, sparsification is more likely to achieve better trade-offs on more "skewed" graphs, i.e., graphs that have many small-degree vertices and few extraordinarily-large degree vertices. In such input graphs, there is not much room to merge subnodes without creating disconnectivity, since those large vertices are not picked as supernodes (according to Algorithm 3.2). However, in practice, most real-world graphs are less skewed and thus more suitable for V-Combiner. For BP, both schemes have similar results. Finally, for CD, sparsification generally gets better results, especially for highly dense graphs such as CD-FS. In contrast,

V-Combiner achieves better results in sparser graphs such as CD-TW.

Comparing V-Combiner to k-core. Figure 3.9 shows that in most experiments, V-Combiner exhibits better performance-accuracy trade-offs than k-core. The exceptions are HITS-FS, HITS-PLD, HITS-AR, and HITS-DB, for the reason indicated before. However, we will see in Section 3.8.2 that the end-to-end time of k-core is significantly affected by the pre-processing overhead.

3.8.2 End-to-End Analysis

Figure 3.10 shows the total execution time of the best end-to-end configurations for the different benchmark-graph pairs and different techniques. For each benchmark-graph pair, we show, from left to right, bars for the exact, V-Combiner, sparsification, and k-core techniques, all normalized to exact. The numbers on top of the bars are the end-to-end speedups over exact. The missing bars (sparsification in HITS-PLD, and k-core in BP-PLD, BP-DB, CD-BD, and PR-BD) are for configurations that failed to reach the threshold accuracy. The accuracy of each experiment was shown in Table 3.4.

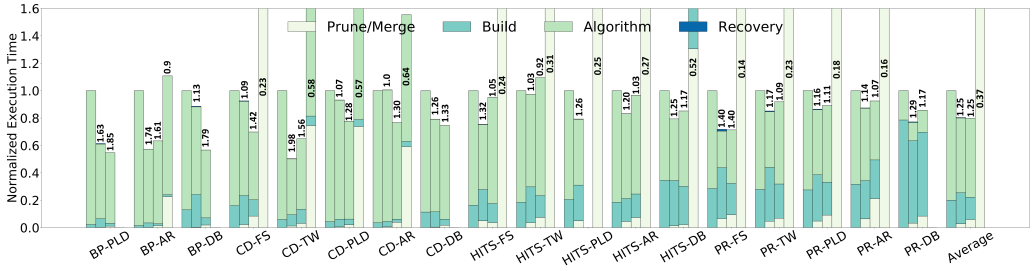


Figure 3.10: Total execution time of the best end-to-end configurations for the different benchmark-graph pairs and different techniques. For each benchmark-graph pair, we show, from left to right, bars for the exact, V-Combiner, sparsification, and k-core techniques, all normalized to exact. The numbers on top of the bars are the end-to-end speedups over exact. The missing bars (one in sparsification and four in k-core) are for configurations that failed to reach the threshold accuracy.

The bars are broken down into the different components of the execution time as shown in the steps of Figure 3.4: (i) pruning or merging, (ii) building the graph, (iii) executing the graph algorithm, and (iv) post-processing or recovering. From the figure, we see that the prune/merge step is cheap

in V-Combiner and sparsification. However, it is very expensive in most of the k-core experiments: even though we use the state-of-the-art k-core implementation [85], pruning until the remaining vertices have a degree of at least k has substantial overhead.

The build step takes, on average, 20%, 22.6%, 16.4%, and (not shown) 19.7% of the *total exact time*, in exact, V-Combiner, sparsification, and k-core, respectively. The build time in V-Combiner is higher than in sparsification because, in this step, V-Combiner performs the actual vertex merging and generates the delta graph. (Recall that, in the merge step, V-Combiner executes Algorithm 3.2, which computes the new vertex degrees). However, the algorithm time is often shorter in V-Combiner than in sparsification, as we saw in Table 3.4. The recovery overheads are negligible.

Overall, we see that V-Combiner obtains an average end-to-end speedup of 1.25× over the baseline across the 18 benchmarks. It does so with an average accuracy of 91.8% (Table 3.4). Sparsification typically has a slower algorithm but a shorter pre-processing time. The resulting average end-to-end speedup of sparsification over baseline is like V-Combiner in Figure 3.10. However, one of the benchmark-graph pairs (HITS-PLD) does not reach the required accuracy in sparsification. Specifically, it can be shown that HITS-PLD only reaches 71% accuracy in sparsification. Such experiment is not included in the average sparsification bar of Figure 3.10. As a result, V-Combiner is preferable over sparsification. Finally, K-core exhibits a substantial slowdown over baseline on average due to its large pruning overhead.

3.8.3 Analysis of the Connectivity

Table 3.5 shows pruning or merging statistics for different techniques with the configurations of Table 3.4. For V-Combiner, we show the breakdown of the vertices in the original graph into supernodes, subnodes, and regular vertices. On average, the fraction of supernodes, subnodes, and regular vertices is 0.1%, 18.5%, and 81.4%, respectively. The percentage of vertices that remain in the approximate graph is equal to the percentage of supernodes plus regular vertices. On average, it is 81.5%. The percentage of edges remaining in the V-Combiner approximate graph is 71.4% on average.

For sparsification, the table shows the percentage of remaining edges (since

Table 3.5: Pruning or merging statistics.

Benchmark & Graph	V-Combiner super	sub	regular	edges	sparsif. edges	k-core vertices	edges
BP-PLD	0.0%	21.6%	78.3%	53.4%	51.7%	-	-
BP-AR	0.1%	37.3%	62.6%	50.6%	53.1%	90.2%	99.6%
BP-DB	0.4%	20.9%	78.7%	61.4%	55.2%	-	-
CD-FS	0.4%	27.6%	72.0%	83.9%	50.0%	84.2%	99.4%
CD-TW	0.0%	7.5%	92.5%	59.1%	50.0%	65.5%	98.8%
CD-PLD	0.0%	4.6%	95.4%	79.0%	68.0%	81.1%	99.2%
CD-AR	0.0%	0.6%	99.3%	99.8%	54.4%	92.7%	99.7%
CD-DB	0.0%	3.7%	96.3%	90.4%	54.3%	-	-
HITS-FS	0.7%	30.6%	68.7%	54.4%	80.0%	24.4%	76.5%
HITS-TW	0.1%	10.4%	89.5%	78.1%	84.8%	22.9%	84.1%
HITS-PLD	0.0%	25.4%	74.6%	75.3%	-	8.1%	55.8%
HITS-AR	0.1%	15.0%	84.9%	86.1%	89.9%	31.7%	75.8%
HITS-DB	0.0%	8.9%	91.1%	87.6%	90.1%	11.8%	65.4%
PR-FS	0.1%	36.6%	62.7%	51.4%	50.0%	41.1%	91.7%
PR-TW	0.1%	21.3%	78.6%	63.0%	80.2%	38.1%	94.1%
PR-PLD	0.0%	19.2%	80.8%	72.3%	70.0%	77.4%	98.9%
PR-AR	0.0%	20.4%	79.6%	82.6%	50.4%	90.2%	99.6%
PR-DB	0.2%	15.6%	84.2%	57.4%	90.1%	-	-
Average	0.1%	18.5%	81.4%	71.4%	66.0%	54.4%	87.6%

no vertex is removed). On average, such number is 66.0%. Finally, for k-core, the table shows the percentages of remaining vertices and edges. On average, they are 54.4% and 87.6%.

V-Combiner keeps more remaining vertices in the approximate graph than k-core. This explains why V-Combiner provides better connectivity than k-core. We also see that, on average, the techniques generate approximate graphs with 65%-90% of the edges, which provide the most profitable performance-accuracy trade-off. Finally, sparsification has the lowest percentage of edges remaining. Because it does not remove vertices, it can tolerate dropping so many edges while still preserving good connectivity.

3.8.4 Analysis of the Average Length of Paths

Table 3.6 compares the average length of the paths in each benchmark-graph pair in the original and approximate graphs. To compute the average path length in each original graph, we select and run 10,000 shortest path queries and average out all the shortest path distances. Next, we use the same queries to measure the average length of the paths in the approximate graphs that the different techniques generated using the configurations of Table 3.4. Finally,

Table 3.6: Average length of the paths in different graphs. The percentage numbers show the percentage of error relative to the original graph.

Benchmark & Graph	original	V-Combiner	sparsification	k-core
BP-PLD	3.73	3.25(12.9%)	4.68(25.5%)	-
BP-AR	7.18	6.18(13.9%)	8.43(17.4%)	4.66(35.1%)
BP-DB	3.81	3.39(11.0%)	5.33(39.9%)	-
CD-FS	5.82	6.13(5.3%)	8.29(42.4%)	4.28(26.5%)
CD-TW	4.21	4.22(0.2%)	5.30(25.9%)	2.19(48.0%)
CD-PLD	4.34	4.36(0.5%)	5.54(27.6%)	0.58(86.6%)
CD-AR	17.64	17.62(0.1%)	21.10(19.6%)	4.38(75.2%)
CD-DB	5.20	5.06(2.7%)	8.63(66.0%)	-
HITS-FS	5.82	5.95(2.2%)	7.73(32.8%)	4.69(19.4%)
HITS-TW	4.21	4.29(1.9%)	4.77(13.3%)	3.15(25.2%)
HITS-PLD	4.34	4.35(0.2%)	-	1.76(59.4%)
HITS-AR	17.64	17.86(1.2%)	18.47(4.7%)	16.86(4.4%)
HITS-DB	5.20	5.10(1.9%)	6.65(27.9%)	4.22(18.9%)
PR-FS	5.82	6.06(4.1%)	8.27(42.1%)	5.09(12.5%)
PR-TW	4.21	4.39(4.3%)	5.30(25.9%)	3.55(15.7%)
PR-PLD	4.34	4.36(0.5%)	5.55(27.9%)	4.23(2.5%)
PR-AR	17.64	18.02(2.2%)	20.80(17.9%)	0.5(97.2%)
PR-DB	5.20	5.05(2.9%)	8.60(55.0%)	-
Mean error	-	3.8%	30.1%	37.6%

we compute the error as the difference between the values in the original and approximate graphs, as a percentage.

V-Combiner preserves the average length of paths much more than sparsification or k-core. On average, the error in average path length in V-Combiner is only 3.8%, while it is 30.1% and 37.6% in sparsification and k-core, respectively. Generally, k-core reduces the average length of the paths, while sparsification increases it. The reason is that k-core prunes vertices and therefore reduces the number of hops to go from one vertex to another. Moreover, k-core often disconnects parts of the graph, causing longer paths to disappear, and the average path length to decrease. In contrast, sparsification increases the number of hops between vertices because it reduces the connections between the vertices by removing edges.

3.8.5 Analysis of Pruning/Merging Parameters

Table 3.7 shows the pruning or merging parameters that we use to generate the best end-to-end configurations for each benchmark-graph pair and technique. In V-Combiner, the parameters are the supernode thresholds α and

Table 3.7: Best parameters of the different techniques.

Benchmark & Graph	V-Combiner α	β	π	sparsification s param	k-core k param
BP-PLD	85%	95%	80%	0.7	-
BP-AR	85%	95%	60%	0.7	2
BP-DB	75%	85%	70%	0.9	-
CD-FS	85%	95%	70%	0.7	2
CD-TW	85%	95%	75%	0.5	5
CD-PLD	85%	95%	20%	0.9	2
CD-AR	85%	95%	5%	0.9	2
CD-DB	85%	95%	20%	0.9	-
HITS-FS	65%	70%	60%	0.7	40
HITS-TW	70%	75%	50%	0.9	25
HITS-PLD	65%	70%	30%	-	40
HITS-AR	65%	70%	20%	0.7	30
HITS-DB	65%	70%	30%	0.7	25
PR-FS	75%	85%	60%	0.7	15
PR-TW	75%	85%	60%	0.5	10
PR-PLD	75%	85%	40%	0.9	2
PR-AR	85%	95%	15%	0.5	2
PR-DB	75%	85%	40%	0.9	-

β, and the subnode threshold π. The table shows these parameters as the corresponding values in the CDF of number of edges (Figure 3.8). We observe that, for CD and BP, the best (α, β) values are mostly (85%, 95%). For HITS, the best values are mostly (65%, 70%), and for PR, they are mostly (75%, 85%). The π parameter shows more variation, as it generally ranges between 60% and 20%. In contrast, for sparsification and k-core, we do not see a clear trend on how to set the s and k parameters; hence a full sweep of parameters is required.

3.9 Related Work

Previous research focused on algorithm-specific approximations such as Frog-Wild [57] for page rank and approximate K-level asynchronous Breadth-First Search [92]. While these approximations are effective for one algorithm, they often lack the generality to be applied to a broad range of algorithms. More general approaches include graph summarization techniques [67, 68, 69, 70, 71, 72, 73], and other graph processing approximate frameworks [63, 64, 55], or general-purpose frameworks that can be applied to graph processing domains too [62, 93]. Compared to the general-purpose

frameworks, V-Combiner is deterministic and provides application transparency.

Graph summarization techniques vary widely from algorithm-specific such as sketching algorithms [68, 69, 70, 71] to more general-purpose such as sparsification [72, 56] and k-core [73]. The original proposal of sparsification [72] provides bounds for the page rank algorithm. However, it is not applicable in practice. First, it requires the graph input to be undirected, while most of the real-world graphs are directed. Second, it requires the knowledge of the graph eigenvalues, which will take much higher time to compute than the actual page rank values. A practical implementation of sparsification [56] replaced the eigenvalues with a tunable sparsification parameter and average degree of the graph that provides end-to-end performance gains, but with no accuracy guarantees.

The idea behind sketching algorithms is to summarize the graph information tailored to a specific application into a data structure that can be queried later to return a fast approximate answer after simple computations. Unfortunately, while sketching techniques provide high accuracy, their end-to-end performance is much worse than the exact baseline due to their high pre-processing overheads. V-Combiner has lower overhead and hence provides higher end-to-end performance at high accuracy. In contrast to some popular sketching algorithms that are tailored to specific graph algorithms, V-Combiner can be applied to a wide variety of graph algorithms, without modifying their code.

3.10 Conclusion

Fast graph processing has an important role in many applications where a small level of inaccuracy is acceptable. To speed-up iterative graph processing algorithms, we propose to merge certain graph vertices into hub vertices next to them, i.e., vertices with many connections. Our novel scheme, *V-Combiner*, systematically and deterministically constructs an approximate graph using pruning and merging. It also includes an inexpensive correction step after the graph algorithm executes, to recover the contribution of the pruned vertices. The result is faster execution at acceptable accuracy levels.

We evaluated V-Combiner on a 44-core shared-memory platform. On av-

erage across four applications and five graph datasets, V-Combiner attained an end-to-end speedup of 1.25× over the exact baseline system, with an accuracy of 91.8%. We also showed that V-Combiner provides an overall better performance-accuracy trade-off than the sparsification and k-core techniques.

CHAPTER 4

KEEPCOMPRESSED: RETAINING A COMPRESSED GRAPH UNDER DYNAMIC UPDATES

4.1 Introduction

An increasing number of graph workloads operate in dynamic environments, where graphs evolve as they undergo updates to their structure. Such updates can add connections or vertices to the graph or delete existing ones. For example, Twitter recommendation graphs are updated frequently, e.g., upon events such as a like or a reply [94].

Figure 4.1 illustrates a dynamic graph at three consecutive snapshots. At time $t+1$, the graph snapshot has incorporated the addition of edges $\{1, 6\}$ and $\{2, 3\}$, and at time $t+2$, the graph has lost edge $\{1, 4\}$. Concurrently, graph queries have arrived that need to process the most recent graph snapshots. Such queries run a specific graph algorithm on the graph and may take several iterations to converge for billion-scale graphs [6, 95, 96], potentially taking minutes to complete. Hence, it is critical to run the graph algorithms efficiently as the graph dynamically changes over time.

Previous work has shown that a way to speed-up the execution of a graph algorithm while maintaining high output accuracy is to use graph compression techniques such as vertex clustering and merging [97, 98, 6, 99, 100]. These techniques generate a compact form of the graph. For example, V-Combiner [6] compresses the graph by creating large clusters of small-degree vertices and reduces the number of vertices and edges that need to be processed. Further, there are minimum description length (MDL) approaches [97, 98, 99] such as SWeG [97] that aim to describe a graph using the minimum number of edges. SWeG creates clusters of vertices with common neighbors in an iterative fashion and only connects such clusters together if the majority of their internal vertices are connected to one another.

It would be beneficial to use these graph compression techniques in dy-

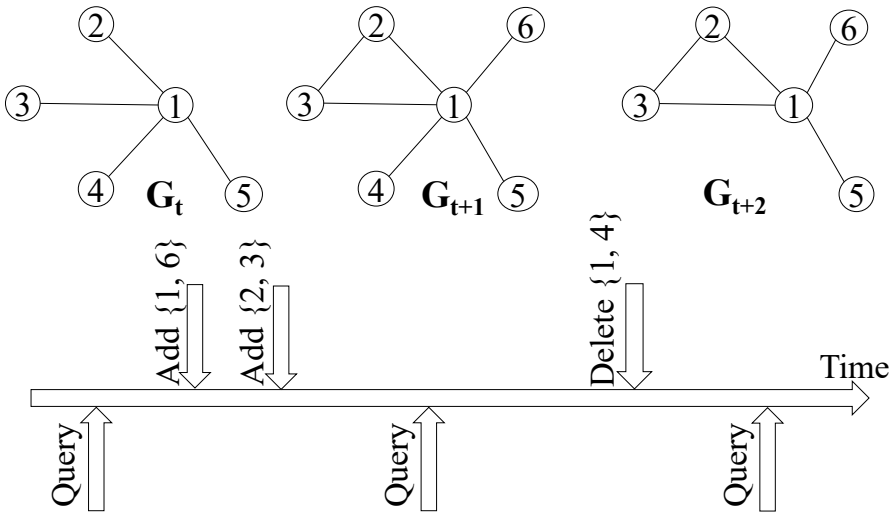


Figure 4.1: Representation of a dynamic graph.

namic graph environments. Unfortunately, it is too expensive to repeatedly regenerate the full graph and then re-compress it every time that new updates arrive. The reason is that tuning the compression parameters, e.g., the number of iterations in SweG [97], or the degree thresholds in V-Combiner [6], has high overhead.

Hence, in this chapter, we propose a new approach: compress the graph only once and minimally adjust it as batches of updates arrive. For this approach to be successful, we need two pieces of information. First, we need to remember which vertices in the original (i.e., full) graph belong to which clusters in the compressed graph. Second, we need to remember the number of existing connections between clusters in the compressed graph. Using such information enables us *to retain the compressed graph under dynamic updates, without having to get the original graph and re-compress it over and over again.*

Unfortunately, state-of-the-art compression via vertex clustering techniques, including V-Combiner and MDL approaches, have major limitations that induce execution overheads and memory consumption. Specifically, V-Combiner requires updating a special graph called delta graph, which it maintains to

be able to compute the values of the internal vertices of clusters at the end of the algorithm run. Moreover, both V-Combiner and MDL-based approaches can potentially create vertex clusters with an unbounded number of internal vertices. Such large clusters can substantially increase the amount of memory consumed to store information on the number of edges between clusters in the compressed graph.

Our contribution. We propose KeepCompressed (KC), a set of algorithms to keep using compressed graphs in a dynamic environment with high performance and low memory overhead. We present a fast parallel vertex clustering algorithm to create clusters of bounded numbers of vertices with similar neighborhoods and substantially reduce the required memory. Further, we drop connections between clusters if the graph is not sufficiently compressed. We leverage the Aspen graph streaming framework [101] and augment it such that we can store information on the number of edges between clusters in the compressed graph. Therefore, we reduce the execution time by running the graph algorithm on the compressed graph while keeping the accuracy high by computing the values of internal vertices based on the values of their clusters.

Results. We evaluate KC for iterative graph algorithms in a dynamic setting across 3 applications and 5 graph datasets. We use a shared-memory NUMA server with 44 cores. Compared to the execution on an uncompressed graph, KC attains an average speedup (including graph update and generation time) equal to 1.22× at an average accuracy of 94.0%, while using only 11.72% additional memory.

4.2 Background

A graph G=(V, E) is represented using a set of nodes or vertices (V) connected via a set of edges (E). It can be *directed* or *undirected* depending on the type of the edges. If an edge does not have a direction, it can be denoted using a set of size two, e.g., $\{u, v\}$. For each vertex, the number of its connections (or neighbors) determines its *degree*. A tree is an undirected graph such that any two nodes are connected by exactly one path. A binary tree is a tree in which every node has at most two children. For clarity, we will use the term vertex for graphs and the term node for trees. A *vertex cluster* is a set of vertices and is denoted using the letters A, B, etc. The size

of a vertex cluster is denoted by $|A|$ and represents the number of vertices inside that cluster. We refer to such vertices as *internal vertices*. A *single vertex* is a cluster of size 1. We will use the term cluster to refer to clusters of size more than 1.

A large number of graph processing and analytic workloads process graphs that dynamically update their structure over time. For example, Twitter maintains and updates recommendation graphs upon events such as like or reply [94]. These *graph updates* to the graph structure happen as a stream of edge modifications of two types: adding a new edge or removing an existing edge. A dynamic graph can be represented as a sequence of graphs G_t, each representing a static *snapshot* of a changing graph at each point in time t. A *graph query* executes a specific graph algorithm on a static snapshot of the graph. Figure 4.1 illustrates how a dynamic graph is represented at different points in time. Queries happen independently from the updates. Updates can be applied to the graph one at a time or in *batches*. In shared-memory systems, batching is necessary to exploit the parallelism of applying the updates [101, 102, 103, 104, 105].

Aspen. Aspen [101] is the state-of-the-art graph data structure for fast streaming of dynamic graph updates. Many data-structures for dynamic graph processing [96, 106, 102, 107] trade-off memory-efficiency for performance. Aspen provides high-performance thanks to memory compression techniques. The main insight of Aspen is to represent a graph as a tree of trees. Both trees are binary trees. Specifically, Aspen includes a 1) Vertex Tree (VT) to store individual vertices of the graph and 2) an Edge Tree (ET) for each vertex of the graph (a node in the VT) to store neighbors of that vertex. This way of representation allows for logarithmic-time addition or deletion of vertices or edges to the graph.

To reduce the overhead of pointer chasing in trees, Aspen sorts neighbors of each vertex and puts neighbors of the same range into a chunk. Chunking also allows for memory compression optimizations by storing the differences of sorted neighbor IDs. Specifically, Aspen uses a randomly-chosen set of vertex IDs as *keys* to specify the beginning of a range of neighbor IDs. Each ET node can then be specified using a pair $<$ key, range $>$. The neighbors' IDs smaller than the first key are not organized in the tree and form the *prefix*. Overall, each ET can be specified using a pair $< P, T >$, where P is the prefix, and T contains the ET nodes.

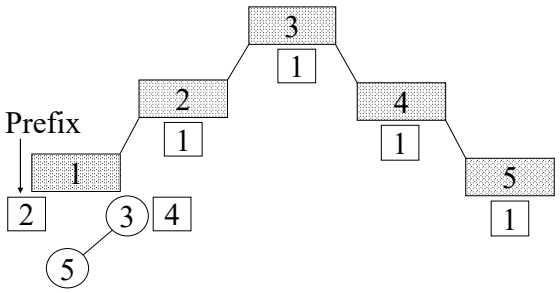


Figure 4.2: Aspen for Figure 4.1 for the first query.

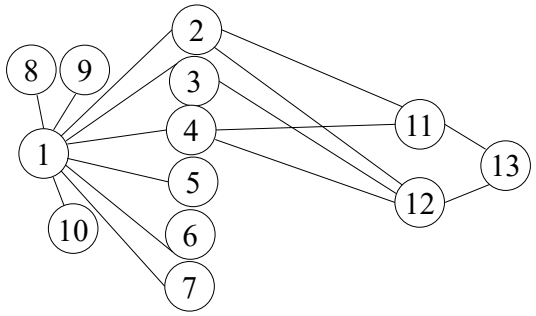


Figure 4.3: Example graph.

Figure 4.2 demonstrates an Aspen representation of the graph of Figure 4.1 at time t. The graph has vertices ① to ⑤ shown in the VT with shaded rectangular nodes. The neighbors of each vertex are shown using an ET at the bottom of each VT node. For example, vertex ① has neighbors ② to ⑤ with ③ and ⑤ being the keys. The prefix contains any neighbor less than ③, i.e. ②, and ④ is included in the range of key ③. The other vertices ② to ⑤ all have only ① as their neighbor, and that constitutes the prefix.

Vertex clustering. Graph compression techniques can be used to create a compact representation of an existing graph. *Vertex clustering* techniques [97, 98] compress the graph by grouping several vertices as one cluster. The result is the reduced number of vertices and edges between them. In addition, the graph algorithm can be executed more efficiently whenever a query arrives. The compression can be either lossless [97, 99], which allows for accurate reconstruction of the original graph, or lossy [98, 6], which loses some of the information, affecting the accuracy of the query results. In this

chapter, we focus only on lossy techniques since they are able to substantially speed-up the graph algorithm on the compressed graph.

Compression criteria. By clustering vertices, a new *compressed* graph $G'=(V', E')$ is created from an original *exact* graph $G=(V, E)$, such that $V' < V$ and $E' < E$. There are two criteria required to define a vertex clustering technique. First, we determine which different vertices in V are eligible to form a new cluster in V'. We can do so by keeping a mapping from V to V'. Note that single vertices are neither a vertex cluster nor internal vertices of a cluster; hence, they are mapped to themselves. Overall, the new vertices in V' are either clusters or single vertices. Second, we decide 1) how edges of E are transformed into E' and 2) whether we want to keep them. Specifically, if the vertex mapping for either u or v in $\{u, v\} \in E$ has changed, we transform the edge using the new mappings for u and v. Indeed, if we use large clusters, many edges in E will be mapped to the same edge in E' or be completely dropped, potentially making E' significantly smaller than E.

V-Combiner [6] (discussed in Chapter 3) uses a technique called "vertex merging" to cluster vertices with small degrees (subnodes) into nearby higher-degree vertices (supernodes). Figure 4.4 shows how V-Combiner performs vertex clustering for the graph of Figure 4.3. Vertex ① is a supernode and can cluster vertices ② to ⑩ into itself. V-Combiner drops the edges between the subnodes ② to ⑩ and the supernode ①, and renames the supernode as a new cluster Ⓐ. It places two edges from Ⓐ to ⑪ and ⑫ in the compressed graph, since at least one subnode was connected to these vertices in the original graph. Finally, since subnodes do not participate in the computation of the graph algorithm, V-Combiner proposes a delta graph to recover the value of subnodes after the graph algorithm executes.

Compression parameters. V-Combiner takes as input a pair of upper and lower degree thresholds (α, β) to choose the supernodes. Specifically, it selects a vertex as a supernode if its degree is between the two thresholds. Additionally, it selects a vertex as a subnode if its degree is below a certain threshold. To reduce the tuning time, V-Combiner provides graph-specific ranges to choose the thresholds.

Minimum description length (MDL) approaches [98, 97, 99] are based on the information-theoretical concept of "minimum description length" to describe the edges in a graph in the most compact form possible. The idea

Algorithm 4.1: MDL Lossy vertex clustering [97].

inputs : vertices V, edges E, iteration i
outputs: Vertices V', Edges E' in the compressed graph

hash: $v \rightarrow \{1, ..., |V|\}$;
for v ***in*** V **do**
    for n *in* `Neighbors` (v) **do**
        hash$[v] \leftarrow \min($hash$[v]$, hash$[n])$
Create histogram of vertices based on their hashed values;
Divide vertices into disjoint groups of $\{g^1, ..., g^k\}$;
$V', E' \leftarrow V, \{\}$;
for g^i ***in*** $\{g^1, ..., g^k\}$ **do**
    while $|g^i| > 1$ **do**
        Pick and remove a random vertex u from g^i;
        $v \leftarrow \text{argmax}_{w \in g^i}$ `Saving` (u, w);
        if `Saving` $(u, v) > (1 + i)^{-1}$ **then**
            ▷ cluster u and v together in A, update V' and g^i
            $V' \leftarrow (V' - \{u, v\}) \cup A$;
for A ***in*** V' **do**
    for B ***in*** V' *s.t.* $\exists \{u, v\} \in E$, $u \in A$, $v \in B$ **do**
        |edges| $\leftarrow$ `NumberOfExistingEdges` (A, B);
        if |edges| $> \frac{|A| \times |B|}{2}$ **then**
            $E' \leftarrow E' \cup \{\{A, B\}\}$;

is to leverage vertex clustering *iteratively* to minimize the number of edges in the compressed graph. The compression criteria are typically based on heuristics and can also be influenced by the target graph processing algorithm. We focus on the heuristics from SWeG [97] for its simplicity and potential application in the iterative graph processing algorithms.

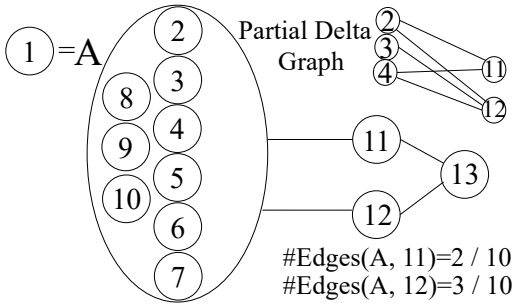
Algorithm 4.1 illustrates vertex clustering in MDL in a single iteration. First, it creates groups of vertices that can potentially be clustered together. Using a technique called MinHashing [108], vertices with similar neighborhoods are grouped together (Lines 1–6). Then, it processes each group in parallel to obtain V'. The idea is to cluster vertices u, $v \in V$ with as many common neighbors as possible to compress the graph by minimizing $|E'|$. The more common neighbors between u and v, the smaller $|E'|$. Figure 4.5 shows how MDL performs vertex clustering for the example of Figure 4.3. Clusters Ⓐ, Ⓑ, and Ⓒ include vertices ②-④, ⑤-⑦, and ⑪-⑫, respectively.

The function *Saving*(u, v) provides a metric for the number of reduced edges in E'. The clustering only succeeds if the *Saving* is above a threshold $(1+i)^{-1}$, which decays with the number of iterations. Finally, for each pair of clusters A, B in V' s.t. $\exists$ $\{u,v\} \in E$, $u \in A$, $v \in B$, we check whether there should be an edge in the compressed graph (Lines 14–18). The function *ExistingEdges*(u,v) returns the number of edges $\in E$ between the internal vertices of u' and v', which can be either clusters or single vertices (cluster of size 1). The number of total possible edges between them is equal to the product of the sizes of the two clusters ($|u'| \times |v'|$). If the number of existing edges is greater than half of the total possible edges, we place an edge $\in E'$ between u' and v'. The compression can run in several iterations by feeding the output of a previous iteration to the next one.

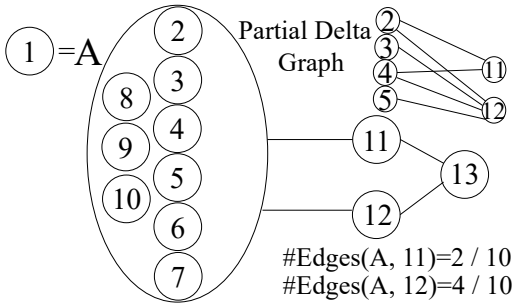
Compression parameters. MDL takes as input the number of iterations to perform Algorithm 4.1. The more iterations we take, the more compressed the graph will become.

4.3 Retaining a Compressed Graph

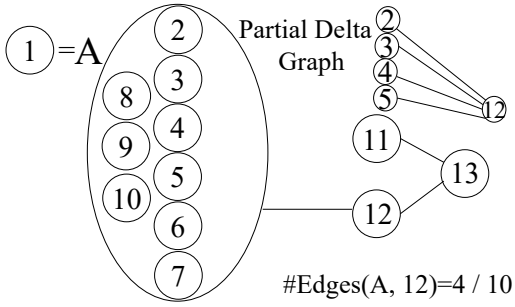
V-Combiner [6] and MDL (i.e., SweG [97]) are two vertex clustering techniques used to process queries on a graph snapshot efficiently. If the graph



(a) G_t, initialize a compressed graph

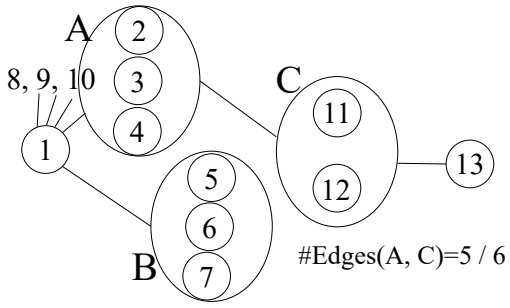


(b) $G_t + 1$, add $\{5, 12\}$

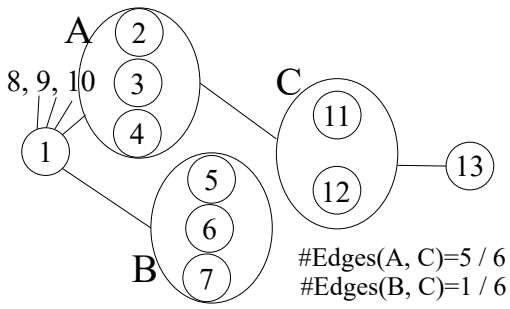


(c) $G_t + 2$, delete$\{2, 11\}$, $\{4, 11\}$

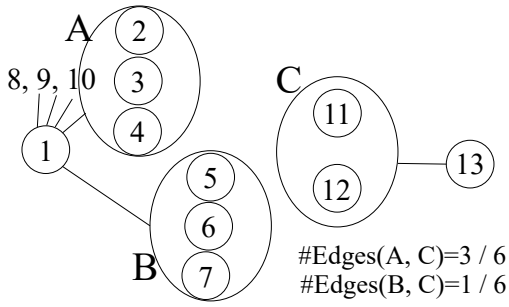
Figure 4.4: Illustration of V-Combiner on the graph of Figure 4.3 under dynamic graph updates.



(a) G_t, initialize a compressed graph



(b) $G_t + 1$, add {5, 12}



(c) $G_t + 2$, delete{2, 11}, {4, 11}

Figure 4.5: Illustration of MDL on the graph of Figure 4.3 under dynamic graph updates.

compression algorithm is fast, it is beneficial to execute the query on the compressed graph snapshot due to the reduced number of vertices and edges. However, in practice, the compression can take longer due to the tuning of the compression parameters required to yield favorable speedups under acceptable accuracy. Therefore, it is not profitable to constantly re-compress a dynamic graph every time that graph updates arrive.

With our proposal, we retain a compressed graph under dynamic graph updates. Hence, we do not need to compress the graph over and over again. Indeed, we assess new graph updates as they arrive and quickly decide how to incorporate them into an existing compressed graph.

Although both V-Combiner and MDL techniques can compress a static graph, neither was initially designed to support dynamic graph updates. In this section, we investigate their potential in an environment that retains the compressed graph. Moreover, we assess whether the overhead of retaining a compressed graph is small enough to result in overall reduced query execution time.

The initial step to retain a compressed graph using any vertex clustering technique is to remember the vertices inside each cluster. Whenever a new edge update $\{u, v\}$ arrives, we can assess whether u and v belong to clusters in V'. In addition, we need to remember how many edges exist between the internal vertices of two vertex clusters or between the internal vertices of a vertex cluster and a single vertex. By remembering these two pieces of information, some new graph updates may not need to update the compressed graph.

Consider the example of Figure 4.3. Figures 4.4 and 4.5 illustrate how each technique could potentially implement the idea of retaining the compressed graph. Figures 4.4 (a) and 4.5 (a) show the initial compressed graphs at time t using V-Combiner and MDL techniques, respectively. For V-Combiner, supernode ① is renamed to cluster Ⓐ and has clustered subnodes ② to ⑩ into itself. We place an edge between Ⓐ and vertices ⑪ and ⑫, since at least one subnode is connected to these vertices. We also show the partial delta graph for subnodes with connections to vertices ⑪ and ⑫. For MDL, vertices ② to ④, ⑤ to ⑦, and ⑪ to ⑫ form three different clusters Ⓐ, Ⓑ, Ⓒ. According to Lines 17–18 of Algorithm 4.1, we place an edge only between Ⓐ and Ⓒ, since 5 out of 6 total edges are present in the original graph.

For both techniques, the expression #Edges(u, v) shows the number of

existing edges and the number of total possible edges between the internal vertices of u and v. Note that at least one of the vertices should be a cluster. For example #Edges(A, C) = 5/6 means that there are five edges (out of six possible edges) between the internal vertices of Ⓐ and Ⓒ.

Next, consider an update at time $t+1$ and two deletions at time $t+2$.

Edge {5, 12} is added at time $t+1$. For V-Combiner, the edge {5, 12} connects a subnode to a vertex. Thus, the delta graph should be directly modified. Moreover, we need to update #Edges(A, 12), since vertex ⑤ belongs to cluster Ⓐ. For MDL, ⑤ and ⑫ belong to two different clusters Ⓑ and Ⓒ. Therefore, we create and set #Edges(B, C) to 1/6.

Edges {2, 11} and {4, 11} are deleted at time $t+2$. For V-Combiner, these edges connect two subnodes to a vertex. Therefore, the delta graph should be directly modified. We also update (and remove) #Edges(A, 11). As there are zero edges left between the internal vertices of Ⓐ and ⑪, we also remove the edge {A, 11} from the compressed graph. For MDL, the deleted edges are between clusters Ⓐ and Ⓒ. The only required change is to record that the number of edges between the two clusters has decreased by 2. Based on the Algorithm 4.1 (Lines 17–18), we remove the edge between Ⓐ and Ⓒ, since #Edges(A, C) indicates that the number of edges is not higher than half of the total possible edges.

Conclusion. Table 4.1 summarizes how well the two techniques can support dynamic updates. Both techniques suffer from limitations, potentially running with high overhead if we want to retain a compressed graph. First, V-Combiner requires additional updates to the delta graph if graph updates are applied to the internal vertices (subnodes). Second, in both techniques, the number of existing edges between two clusters of a compressed graph (as given by the value of #Edges(u, v)) is unbounded.[1] For V-Combiner, supernodes are the larger degree vertices causing cluster sizes to be as large as their degrees. For MDL, recall that vertices inside a group of unbounded size are clustered together. The unbounded group sizes, along with the iterative clustering can lead to the creation of very large clusters. As a result, the amount of memory for storing edge information can increase substantially for both techniques.

[1]From now on, when we use #Edges(u, v), we implicitly mean only the actual number of edges.

Table 4.1: Assessment of existing clustering techniques to support dynamic updates.

Technique	Additional Updates?	Clusters Bounded?
V-Combiner [6]	Yes, to the delta graph.	No, due to large supernodes
MDL [97]	No.	No, due to iterative clustering & unbounded groups

4.4 Overview of KC

Main Idea. Based on our assessment of Section 4.3, we propose a technique called KC (KeepCompressed), which enables retaining compressed graphs under dynamic graph updates at low overhead while being memory-efficient. Therefore, we can continue performing the graph algorithm on the compressed graph by minimally adjusting the graph under the new updates, *without having to re-compress over and over again*. The idea is to create clusters of vertices with *bounded* number of internal vertices. As a result, the information on the number of edges between clusters (or a cluster and a single vertex) is also bounded. Hence, we can store such information efficiently. Further, if the graph is not sufficiently compressed, we drop connections given a certain parameter.

Reference Graph. We use a dynamic data structure named reference graph to store and modify the information on the number of edges efficiently. We will refer to this information as *states*. Using reference graph can regenerate a compressed graph upon new queries or whenever a substantial amount of updates have been applied to the graph. The reference graph is a Vertex Tree (VT) of Edge Trees (ETs) based on Aspen [101]. It is augmented with constant-size state information for the vertices that are neighbors to clusters, and *vice versa*. The VT contains vertices of the compressed graph (V'). Each ET node contains clusters (or single vertices) that could become neighbors to a specific vertex in VT. More precisely, the reference graph contains a neighbor v' for vertex u', if at least one edge $\in E'$ exists between the internal vertices of the two, i.e. #Edges(u', v') > 0.

Processing Model. We envision KC to be deployed in a streaming scenario of dynamic graph updates similar to the one used in Aspen [101]. The difference is that Aspen handles graph updates exactly as they appear, while KC handles them as if they were applied to the compressed graph.

The process starts off by streaming in an initial set of edges from a certain graph. Using the initial set, we create vertex clusters of bounded size and initialize the cluster states. As the graph update batches are streamed, we continue updating the states of existing neighbors, removing existing neighbors, or adding new neighbors. Upon each query, we can quickly generate a compressed graph snapshot using the most recent cluster states. We can then run the graph algorithm on the compressed graph and correct the results of the algorithm for the internal vertices of clusters.

Algorithm 4.2: Vertex clustering algorithm in KC.

inputs : vertices V, edges E, KC parameter *chunk*
output: Mapping of vertices from V to V' (clusterOf), V'

1 Do lines 1–5 of Algorithm 4.1
  ▷ count is the histogram of vertices based on MinHash
2 **for** v ***in*** V **do**
3 Create $\frac{\mathsf{count}[\mathsf{v}]}{chunk}$ groups of maximum *chunk* size;
4 $V' \leftarrow \{\}$
5 **for** g^i ***in*** $\{g^1, ..., g^k\}$ **do**
6 Create a new vertex cluster A;
7 $V' \leftarrow V' \cup A$;
8 **for** v ***in*** g^i **do**
9 clusterOf[v] = A;

4.4.1 Vertex Clustering in KC

The first step in KC is to place vertices with similar neighborhoods into clusters of bounded size. Such clusters are vertices for a new compressed graph, which is a compact form of the exact graph under dynamic updates.

Algorithm 4.2 shows how we perform vertex clustering in KC. Here, we only provide the algorithm for undirected graphs, which one can extend to directed graphs. We leverage Algorithm 4.1 and adjust it such that we have clusters of bounded size. The first is to compute MinHash values of vertices and a histogram of vertices based on their hashed values (Lines 1–5 of Algorithm 4.1). After computing the *count* of vertices with the same hashed value, we can divide similar vertices into disjoint groups of maximum size *chunk* (Lines 2–3). Then, for each group in parallel we create a new

vertex cluster $A \in V'$ and set the cluster of the group's internal vertices as A (Lines 5–9). The array *clusterOf* keeps the mapping of vertices from V to V'. Note that single vertices are mapped to the same ID in V'.

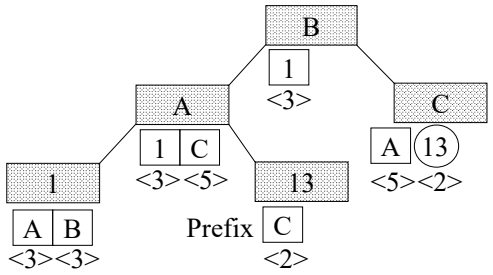
Comparison with MDL Vertex Clustering. Our vertex clustering algorithm has three main differences from the MDL technique [97] to accomplish effective compression of a graph under dynamic updates. Previously, we introduced the *chunk* parameter as an upper bound on the number of vertices of each cluster. However, having clusters with a limited size may hinder sufficient compression in a graph. Therefore, we introduce two more modifications. First, we enforce *ALL* vertices in one group to form a cluster, contrary to MDL, where only some vertices in each group form a cluster, based on a saving score. Second, we introduce a *threshold* parameter $\in$ (0, 1), which forces the graph to drop edges between clusters if the graph is not sufficiently compressed. Formally, we drop an edge between u and v (u, $v \in V'$) if $\#\text{Edges}(u, v) < \text{ceil}(|u| \times |v| \times \textit{threshold})$. Finally, we only run vertex clustering for a single iteration to reduce the compression time, unlike MDL, which needs to run for several iterations for effective compression.

4.4.2 Operations on the Reference Graph

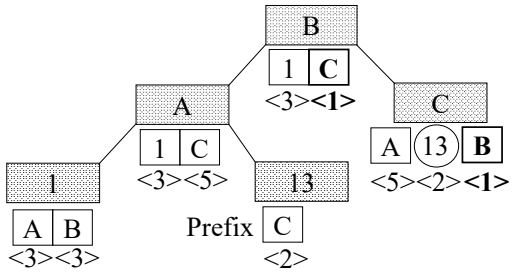
After clustering vertices, we can build an initial reference graph based on Aspen [101] and stream batches of updates. We now explain how we leverage Aspen to support reference graphs for undirected graphs.

1. Storing cluster states. To store the cluster states, we *do not* need additional memory. We can use a few least-significant bits of a neighbor ID and leave the more significant bits to store the ID itself. Using least-significant bits does not change the sorted order of neighbor IDs in Aspen, which is critical for memory compression optimizations.

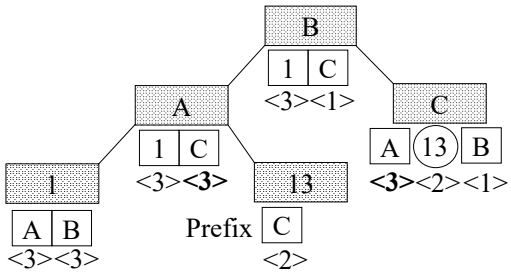
2. Initialization. Algorithm 4.3 presents the initialization of the reference graph. It takes as input the output of KC vertex clustering (Algorithm 4.2) as well as a set of edges E' and KC parameter *threshold*. Using the mapping of vertices from V to V', it transforms the initial set of edges of the original exact graph E into the edges of the compressed graph E' (Lines 1–4). The E' now has many duplicate edges due to multiple vertices in V mapping to the same cluster in V'. The number of duplicate edges initializes the state



(a) Initializing reference graph at t. Vertices ordered as ①, Ⓐ, ⑬, Ⓑ, Ⓒ. ⑬ is a key



(b) Adding $\{5, 12\}$ at $t + 1$



(c) Deleting$\{2, 11\}$, $\{4, 11\}$ at $t + 2$

Figure 4.6: Illustration of reference graph on the example of Figure 4.5 under dynamic graph updates.

between two clusters (or a cluster and a single vertex). After sorting E', we can remove the duplicates and store their count for every neighbor $v \in \{u, v\} \in E'$ using the least significant bits. We slightly modify Aspen [101] to initialize a reference graph given V', E' and *threshold*. *Threshold* determines whether we need to drop the edge between u and v in the compressed graph snapshot.

Figure 4.6 (a) shows the initial reference graph for the example of Figure 4.5 (a). The shaded rectangular boxes are VT nodes and represent vertices in the compressed graph. The vertices are either clusters or single vertices and ordered as ①, Ⓐ, ⑬, Ⓑ, Ⓒ. The square boxes or circles below VT nodes are neighbors organized either as a single prefix (for all vertices except C), or an ET with a prefix (only for cluster C). The circle ⑬ is the only key and specifies the beginning of a range of neighbors in the ET node of cluster C. The values inside ¡¿ are the cluster states and show the number of edges between the internal vertices of a vertex and its neighbor at time t.

Algorithm 4.3: Initializing the reference graph.

inputs : New vertices V', edges E, mapping of vertices from V to V' (clusterOf), KC parameter *threshold*
output: Reference graph Ref initialized with V', E'

```
1 E' ← E;
2 for e in E' do
3     e.src = clusterOf[e.src];
4     e.dst = clusterOf[e.dst];
5 Sort (E');
6 RemoveAndCountDuplicates (E');
7 Ref = InitializeReferenceGraph (V', E', threshold);
```

3. Applying updates. Similar to Lines 1–6 of Algorithm 4.3, we first transform updates from U to U' by removing duplicates and storing their count in the least-significant bits of neighbor IDs. Algorithm 4.4 takes the transformed update batch U' and in parallel generate new ETs $< P_1, T_1 >$ for each vertex w under updates (Lines 3–4). Recall that P_1 is the prefix and T_1 contains the ET nodes. Next, for each vertex w it merges the new ET with its corresponding ET $< Ref.\text{prefixes}[w], Ref.\text{trees}[w] >$ in the reference graph by calling the function *RecursiveUnion*. The idea is to recursively merge the left and right subtrees of each new ET with the left and right subtrees of

their corresponding ETs in the reference graph. The function *BaseUnion* is the base case for this recursive operation which merges a prefix P_1 with an ET $< P_2, T_2 >$. If all the neighbors in P_1 are smaller than the smallest key of T_2, we merge P_1 into P_2 using a function *UnionRange*. However, if some of the neighbors in P_1 are greater than the smallest key of T_2, we need to merge those neighbors in the ranges of their corresponding keys in T_2. Therefore, we split the range of P_1 using the smallest key of T_2 to P_L and P_R. Then, we look for the corresponding keys for the neighbors in P_R using *FindRangesInTree* (Lines 7–9). Using the keys, we can build new ET nodes by creating ranges of neighbors from P_R and merging them into their corresponding ranges of T_2 using *UnionRange* (Lines 10–13). We can then return a new ET using the new ET nodes and the prefix.

Changes to Aspen. The *UnionRange* in KC behaves differently from Aspen when handling updates to existing entries of an ET node ¡key, range¿. In Aspen, if an existing neighbor is updated, it will be either ignored (in case of an addition) or immediately removed (in case of a deletion). In KC, updates to an existing neighbor could result in three cases. In case of additions, we increment the state by the number of duplicate edges. In case of deletions, we either 1) decrement the state by the number of duplicate edges or 2) remove the neighbor if the state reaches 0.

Figures 4.6 (b) and 4.6 (c) illustrate how KC applies the edge updates to the reference graph and modifies the cluster states. At time $t + 1$, edge $\{5, 12\}$ is added to the original graph which is transformed into $\{B, C\}$ of the compressed graph. Since no such connection existed between clusters Ⓑ and Ⓒ, we add new neighbors to the ETs of Ⓑ and Ⓑ (twice since this is an undirected graph). For vertex Ⓑ, neighbor Ⓒ is merged into the prefix of the existing ET. For vertex Ⓒ, Ⓑ joins the range under the existing key 13. Note that Ⓑ comes after Ⓑ in vertex ordering. Since only one edge is added between the two clusters, the initial states are 1. At time $t + 2$, for vertex Ⓐ, we decrement the state of the existing neighbor Ⓒ in the prefix by 2. Likewise, for vertex Ⓒ, we decrement the state of the existing neighbor Ⓐ in the prefix by 2.

4. Generating a graph snapshot. By retaining reference graph under dynamic updates, we can quickly generate a compressed graph snapshot at any time given KC parameter *threshold*, a similar process to *flat snapshotting*

in Aspen. Aspen keeps the total number of edges in each ET at the root of the tree to build an exact graph. Similarly, we store the number of edges in the compressed graph (along with the total number of edges) to generate a compressed snapshot.

Algorithm 4.4: Applying graph updates to the reference graph using Aspen [101].

inputs : vertices V, edge updates U', reference graph Ref, *threshold*
output: A new reference graph reflecting the updates

```
1  W = FindVerticesUnderUpdates (U')
2  for w in W do
3      z = FindEdgesOfVertex (w, U');
4      Create ET < P_1, T_1 > given z and threshold;
5      <prefixes[w], trees[w]> = RecursiveUnion (< P_1, T_1 >,
         <Ref.prefixes[w], Ref.trees[w]>);
6  Function BaseUnion (Prefix P_1, ET < P_2,T_2 >)
7      P_L, P_R = SplitRange (P_1, SmallestKey (T_2));
8      P = UnionRange (P_L, P_2);
9      < keys, ranges > = FindRangesInTree (P_R, T_2);
10     r = CreateEdgeRanges (P_R, keys, threshold);
11     for k in keys do
12         s = UnionRange (r[k], ranges[k]);
13         nodes[k] = < k, s >;
       ▷ Return a new ET using P and nodes
```

4.4.3 Correcting the Results of a Query

When a query arrives, we run the graph algorithm on the most recent version of the compressed graph. The internal vertices of clusters do not participate in the computation. We propose a small post-processing step (similar to the recovery algorithm of V-Combiner [6]) to assign values to the internal vertices based on the values of their clusters. By clustering several vertices as one vertex, we can treat connections to each cluster as connections to individual vertices of that cluster. Therefore, all internal vertices have identical neighborhoods to their cluster, and we can assign the values computed for the clusters to their internal vertices. We can similarly improve the values of internal vertices further by running an iteration of the graph algorithm only for the internal vertices using the values of vertices in the original graph, which could be either internal vertices or single vertices.

4.5 Experimental Setup

We develop KC in C++ using OpenMP to implement clustering and extend Aspen to support dynamic updates on the reference graph. We use GAP [4] interface to run benchmarks.

Machine specifications. We run our experiments on a shared-memory system with 192GB of memory in 4 NUMA nodes and 44 Intel Xeon Gold 6152 cores. We use *numactl interleave all* command to average out NUMA latency effects.

Benchmarks and Datasets. We choose our graph datasets from different sources (Table 4.3) and obtain both directed and undirected variants of them. We evaluate three iterative graph processing algorithms since they can tolerate lossy compression in graphs. These algorithms are similar as they compute a value for each vertex by processing all the edges in the graph and running until a convergence metric has been met [6]. Table 4.2 shows the benchmarks and the metrics to measure the accuracy of their results given compressed graphs. We could not run belief propgation on our largest graph dataset (TW) due to memory limits.

Table 4.2: Benchmarks and accuracy metrics.

Benchmark	Graph	Accuracy Metric
Belief propagation (BP) [51]	Undirected	Top-k [57, 6]
Community detection (CD) [52]	Directed	Classification [6]
HITS [54]	Directed	Top-k

Table 4.3: Graph datasets.

Graph dataset	Vertices	Directed Edges	Undirected Edges
Gap-Twitter (TW) [81]	61.5M	1456.1M	-
Soc-Twitter-2010 (ST) [109]	21.3M	265.0M	265.0M
Dbpedia (DB) [84]	18.3M	136.5M	126.9M
Indochina-2004 (IN) [86]	7.4M	191.6M	151.0M
Web-Google (WG) [81]	0.9M	5.1M	4.3M

Accuracy metrics. We follow the accuracy metrics introduced by previous work [57, 6]. The *top-k* accuracy metric computes the ratio of vertices which are included in the top ranking of the results given the exact graph and exist

in the KC results given the compressed graph. The *classification accuracy* (specific to the CD benchmark) measures the percentage of vertices that have been assigned their correct labels using KC.

Execution configurations. We compare the execution times of two different configurations:

• ***Exact***. This is the baseline using Aspen with compression parameter 256. The dynamic graph updates are applied to the Aspen graph as-is. The graph algorithm also runs on the exact graph. The results always have 100% accuracy.

• KC. This is our proposed configuration that also uses Aspen with the same compression parameter. However, the updates are transformed and applied to a reference graph. The graph algorithm runs on the compressed graph, which can be generated at any time from the reference graph. The results have an accuracy threshold of 90%, which is a common threshold used by past work [87, 65, 6]. To set KC parameters in Algorithms 4.2–4.4, we examine combinations of *threshold* $\in \{0.1, 0.25, 0.5, 0.75, 0.9\}$ and *chunk* $\in \{2, 3, 4, 5, 6\}$ and choose the one that yields the highest speedup on the first query. We use the same *threshold* to apply updates to the reference graph for the subsequent queries.

Execution time breakdown. We divide the total execution time into five categories:

• ***Clustering.*** Identifies clusters and internal vertices.

• ***Initialization.*** Initializes the Aspen/reference graph

• ***Updates.*** Applies the updates to Aspen/reference graph

• ***Generation.*** Generates a snapshot from Aspen/reference.

• ***Algorithm.*** Executes the graph algorithm, and corrects its results (only for KC).

Obtaining update batches. We first generate a few previous snapshots of a given graph dataset, as if it were the last snapshot over a series of time units. Specifically, we start with the full graph and collect samples from it in multiple consecutive steps until we are left with 50% of the original vertices. We use random vertex sampling [110], as it is highly effective to capture evolving patterns of graph datasets. Based on the generated snapshots, we obtain the updates to build those snapshots. Overall, we obtain 10 batches of updates for each graph dataset.

4.6 Evaluation

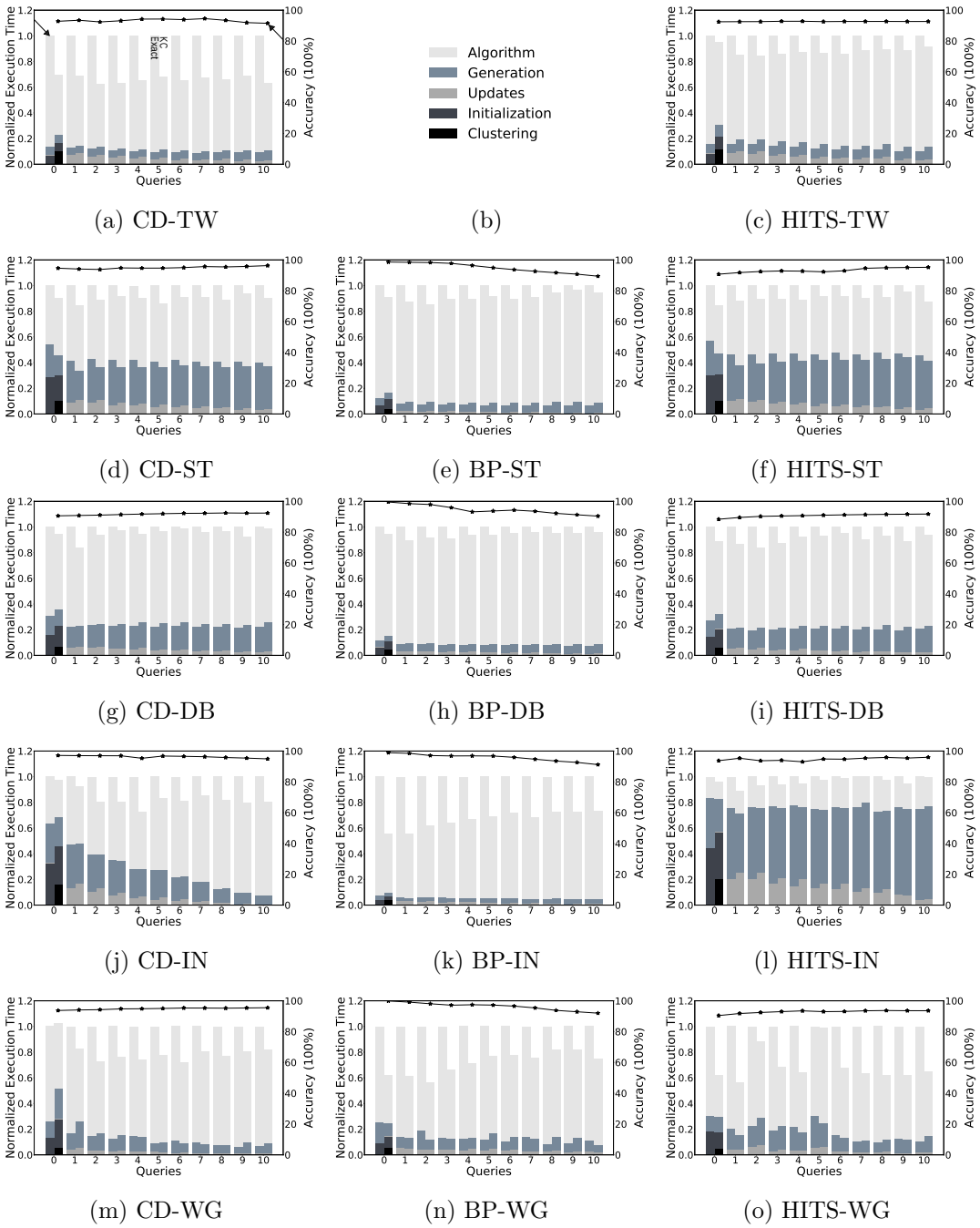
We evaluate both exact and KC configurations in a dynamic setting. In Section 4.6.1, we compare the execution times of consecutive queries of KC and exact. In Section 4.6.2, we compare the graph size in memory and the memory overhead of KC compared to exact. In Section 4.6.3, we present the statistics of the clustering algorithm in KC. Finally, in Section 4.6.4, we compare KC with V-Combiner for retaining a compressed graph.

In the first two sections, as well as the final section, we start by the smallest snapshot of a graph dataset consisting of 50% of the vertices and initialize the reference/Aspen graphs. Next, we stream in 10 different substantially large update batches of varying sizes. After every update batch, we generate a graph snapshot and run the graph algorithm. For KC, we use the best parameters *threshold* and *chunk* for each pair of benchmark and graph, as discussed in Section 4.5. Overall, we evaluate both configurations for 11 consecutive queries.

4.6.1 Analysis of Query times and Accuracy

Figure 4.7 presents the breakdown of query execution times of exact (first bar) and KC (second bar) and their accuracy. The y-axis on the left shows the normalized execution time bars while the y-axis on the right shows the accuracy percentage line. We observe that for all benchmarks except HITS-IN, KC is consistently faster than exact across queries 1 to 10. Notably, our largest graph dataset benchmarks (CD-TW and HITS-TW) achieve monotonously high speedups across queries 1 to 10. This suggests that for sufficiently large graph datasets, KC is most effective in reducing execution time while retaining the compressed graph under substantially large updates. We also observe slight slowdowns in the first query due to the increased overhead of clustering in one benchmark (CD-WG). Unlike HITS and CD, the accuracy slightly drops in BP benchmarks (running undirected graphs). The reason is that undirected graphs can get compressed more aggressively compared to their directed counterparts. In all benchmarks, accuracy remains consistently above 90% threshold for all queries, thanks to the fact that vertices with *similar neighborhoods* are clustered together.

Table 4.4 shows the results of Figure 4.7 averaged across queries 1-10. For each benchmark, we report the breakdown of speedups of KC over exact



(a) CD-TW (b) (c) HITS-TW

(d) CD-ST (e) BP-ST (f) HITS-ST

(g) CD-DB (h) BP-DB (i) HITS-DB

(j) CD-IN (k) BP-IN (l) HITS-IN

(m) CD-WG (n) BP-WG (o) HITS-WG

Figure 4.7: Comparing execution times (left Y-axis) and accuracy (right Y-axis) in exact (first bar) and KC (second bar) configurations.

Table 4.4: Speedup breakdown in exact and KC approaches for queries 1–10.

Benchmark	Speedup($\overline{sp}$)	Update	Generation	Overall	Accuracy(%)
BP-ST	1.14 (1.07)	0.8	0.74	1.1	94.41
BP-DB	1.09 (1.11)	0.84	0.88	1.07	94.1
BP-IN	1.53 (1.6)	0.89	1.06	1.49	95.41
BP-WG	1.5 (1.83)	0.92	1.06	1.4	95.95
CD-TW	1.67 (1.87)	0.77	0.96	1.52	93.28
CD-ST	1.11 (1.34)	0.79	1.22	1.12	95.04
CD-DB	1.11 (1.22)	0.84	0.92	1.05	91.87
CD-IN	1.34 (1.52)	0.76	1.06	1.23	96.21
CD-WG	1.38 (1.53)	0.99	0.93	1.29	94.96
HITS-TW	1.23 (1.24)	0.79	0.77	1.14	92.68
HITS-ST	1.09 (1.29)	0.81	1.21	1.11	93.5
HITS-DB	1.18 (1.26)	0.84	0.91	1.11	90.95
HITS-IN	1.2 (1.37)	0.81	1.07	1.04	94.83
HITS-WG	1.62 (1.94)	0.84	1.07	1.47	93.1
Average	1.3(1.44)	0.84	1.0	1.22	94.0

into algorithm, update, and generation categories. The number denoted by $\overline{sp}$ next to each algorithm speedup is the "expected" speedup. It is computed as $\frac{100}{work}$, where work is the normalized product of the number of iterations and the number of edges in the compressed graph.

In most cases, the actual algorithm speedup is within 15% of the expected speedup, and the average expected speedup is only 10% more than the actual speedup of the algorithm time. The slight variations are due to load-balancing effects and the amount of time we spend to correct the results. Considering only the algorithm time, KC achieves an average speedup of 1.3×. The average generation time is comparable in both KC and exact. The average update time is 16% slower in KC, which is due to the extra time to transform updates from the original to the compressed graph. Overall, we observe a speedup of 1.22× at an average accuracy of 94.0%.

4.6.2 Memory Overhead

Table 4.5 compares the size of the Aspen and reference graphs in memory in the first and last queries. Compared to the first query, the size of the graph in the last query increases by 40.6% and 49.1% using exact and KC, respectively, suggesting that the overall size of updates is substantially large.

Table 4.5: Comparing graph size (GB) in memory.

Benchmark &	Exact		KC		Overhead (%)	
Graph	Q 0	Q 10	Q 0	Q10	Q 0	Q 10
BP-ST	1.58	2.75	1.71	3.43	7.78	24.48
BP-DB	1.25	2.0	1.26	2.33	0.76	16.69
BP-IN	0.57	1.01	0.53	1.02	**-6.73**	1.05
BP-WG	0.06	0.09	0.06	0.09	**-11.02**	**-4.29**
CD-TW	6.46	9.22	7.21	10.73	11.68	16.33
CD-ST	1.63	2.0	1.86	2.41	14.13	20.58
CD-DB	1.3	1.56	1.43	1.8	9.71	15.13
CD-IN	0.6	0.79	0.58	0.82	**-2.51**	3.16
CD-WG	0.07	0.08	0.07	0.08	3.15	6.73
HITS-TW	6.46	9.22	7.34	10.92	13.73	18.38
HITS-ST	1.63	2.0	1.86	2.41	14.01	20.45
HITS-DB	1.3	1.56	1.43	1.8	9.71	15.12
HITS-IN	0.6	0.79	0.58	0.82	**-2.27**	3.6
HITS-WG	0.07	0.08	0.07	0.08	3.12	6.72
Average	1.68	2.37	1.86	2.77	4.66	11.72

We also report the memory overhead of KC compared to exact in the first and last queries. Notably, KC uses less memory than exact in a few benchmarks such as BP-WG. The higher overhead trend in KC is because it stores cluster states in the least-significant bits of neighbor IDs, potentially decreasing chances of compression based on consecutive differences of neighbor IDs. Also, KC consumes additional memory to remember the number of edges in the compressed graph. Compared to exact, KC has only 11.72% memory overhead at the final query.

4.6.3 Clustering Statistics

Table 4.6 shows statistics of the clustering algorithm in KC (Algorithm 4.2). Specifically, we report the number of clusters, the number of internal vertices, average cluster size, and KC parameters *chunk* and *threshold*. On average, the number of clusters is 5.69% of the total number of vertices present in the first snapshot. Therefore, many updates continue to affect non-cluster vertices, helping KC to retain the compressed graph longer. Also, internal vertices are only 13.6% of the total vertices. Therefore, 86.4% of the vertices are still present in the compressed graph, which maintains high accuracy.

For BP benchmarks, the number of internal vertices is 28.88% of the total

Table 4.6: Clustering statistics of KC.

Benchmark	#clusters(%)	#internals(%)	cluster size	*chunk*	*threshold*
BP-ST	11.18	22.37	2.0	2	0.1
BP-DB	9.64	25.97	2.7	3	0.1
BP-IN	17.37	34.73	2.0	2	0.1
BP-WG	8.73	32.44	3.72	6	0.1
CD-TW	1.96	5.98	3.05	6	0.9
CD-ST	0.34	0.79	2.34	6	0.9
CD-DB	0.63	1.94	3.08	6	0.9
CD-IN	9.77	19.53	2.0	2	0.1
CD-WG	3.58	10.04	2.81	6	0.9
HITS-TW	2.14	4.28	2.0	2	0.25
HITS-ST	0.34	0.8	2.33	6	0.9
HITS-DB	0.63	1.94	3.08	6	0.9
HITS-IN	9.79	19.59	2.0	2	0.1
HITS-WG	3.58	10.05	2.81	6	0.1
Average	5.69	13.6	2.57	-	-

vertices, leaving only 71.12% of the vertices in the compressed graph. This observation could explain why undirected graphs are affected more aggressively by the compression. Additionally, the number of clusters is 11.73% of the total vertices of the first snapshot. As a result, fewer updates are likely to affect non-cluster vertices, potentially causing accuracy to degrade over time for BP benchmarks slightly.

We observe that the average cluster size is only 2.77, suggesting that limiting *chunk* threshold is a reasonable design choice. Indeed, for benchmarks with *chunk* of 6, the average cluster size is at most 3.72. Table 4.6 indicates that for BP and CD benchmarks, choosing *threshold* parameter depends on the type of the graph algorithm, rather than the input graph.

4.6.4 Comparison with V-Combiner

In Section 4.3, we stated the performance limitations of the V-Combiner in retaining a compressed graph, particularly the performance overhead of updating the delta graph as well as the significant memory overhead due to the substantially large clusters. Due to these limitations, the overhead of retaining the compressed graph using V-Combiner may adversely affect the speedup of the overall execution. Hence, we only compare the algorithm

Table 4.7: Comparing V-Combiner and KC for the first query.

Benchmark & Graph	V-Combiner		KC	
	Speedup	Accuracy	Speedup	Accuracy
BP-ST	1.55	98.67	1.18	98.65
BP-DB	1.63	99.7	1.12	99.56
BP-IN	1.33	99.17	2.0	98.94
BP-WG	1.61	99.5	1.99	99.93
CD-TW	1.33	92.58	1.85	92.81
CD-ST	1.06	90.52	1.04	94.64
CD-DB	1.24	90.73	1.18	90.6
CD-IN	1.28	95.65	1.27	97.16
CD-WG	1.32	90.32	1.44	93.71
HITS-TW	1.18	93.6	1.31	92.48
HITS-ST	1.09	92.02	1.16	90.75
HITS-DB	1.45	90.42	1.29	88.37
HITS-IN	1.17	91.17	1.23	93.76
HITS-WG	3.3	90.56	2.14	90.41
Average	1.47	93.9	1.44	94.56

speedups of both V-Combiner and KC approaches, normalized to the exact baseline. Table 4.7 illustrates the speedups and accuracy of both V-Combiner and KC approaches for their first queries. We have tuned both approaches for the best speedups above the accuracy threshold of 90%. We observe that averaged across all applications and graphs, V-Combiner and KC attain speedups of 1.47× and 1.44× at an average accuracy of 93.9% and 94.56%, respectively. Therefore, both approaches are comparable, with respect to the first query.

Furthermore, Figure 4.8 depicts the accuracy of V-Combiner and KC, averaged across all applications and graphs, for all the queries. The dashed line shows the accuracy threshold of 90%. We observe that both techniques start with an average accuracy of above 94%; however, the average accuracy of V-Combiner diminishes at a higher rate than that of KC. Particularly, from query 4 onwards, the average accuracy of V-Combiner plunges below the 90% threshold. We can attribute such behavior to the fact that V-Combiner creates substantially larger clusters than KC since it uses high-degree vertices to create clusters of small-degree vertices. As a result, updates are more likely to end up *inside* the clusters in V-Combiner than in KC. Recall from Section 4.3 that we are only able to retain a graph by remembering the state

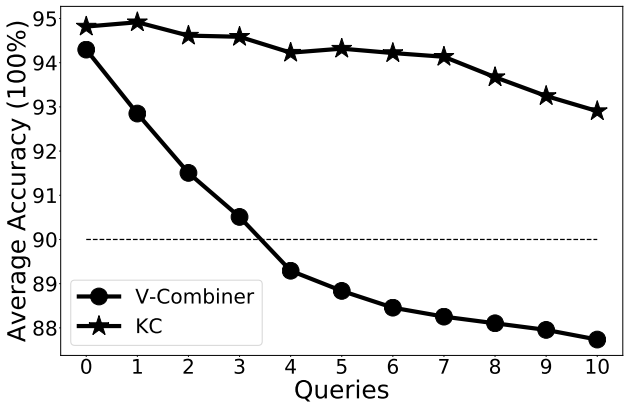


Figure 4.8: Comparing the accuracy of V-Combiner and KC for all queries.

of edges *between* the clusters, rather than *inside* them. For these reasons, V-Combiner retains a compressed graph for a shorter amount of time (fewer number of queries) compared to KC.

4.7 Conclusion

We put forth the idea of retaining a compressed graph to incorporate dynamic graph updates with high performance and low memory overheads. We proposed KeepCompressed (KC) to address the limitations of prior work. KC creates clusters of a bounded number of vertices with similar neighborhoods and drops edges between clusters if necessary. Given bounded cluster sizes, we can retain the information on the number of edges between clusters of a graph at negligible memory overhead and quickly regenerate a snapshot at any time. Therefore, the graph algorithm can continuously utilize a compressed graph under a stream of update batches. Compared to the state-of-the-art streaming graph processing system Aspen, KC attains an overall speedup of 1.22× at an average accuracy of 94%, while using only 11.72% additional memory.

CHAPTER 5

CONCLUSION AND FUTURE WORK

5.1 Conclusion

In this dissertation, we proposed novel software and hardware optimizations to speed-up graph processing workloads. We focused on both categories of data-driven and topology-driven graph algorithms and put forth several optimizations for the compute and scheduling times, without hurting the pre-processing time. Notably, we observed that using approximate and compressed graphs, we can substantially reduce the execution times of many graph processing workloads.

We presented Snug [5] to alleviate the synchronization hotspot that exists in many priority-based task scheduling graph algorithms, without introducing excessive amounts of wasted work. The Snug architecture relies on distributed concurrent subqueues in software and possesses a set of work registers in hardware that point to the highest-priority task in each subqueue. Additionally, it provides an instruction to pick a high-quality task to execute in software that is adaptive to the rate of synchronization failures while consuming acceptable network traffic.

V-Combiner [6] is a software technique to omit unimportant graph vertices from the computation to optimize the compute time. The idea is to identify and prune the unimportant vertices and only retain those with a high number of connections. Additionally, V-Combiner introduces an inexpensive correction step after the graph algorithm executes, to recover the contribution of the pruned vertices. V-Combiner has multiple advantages over traditional general-purpose approximations, including deterministic behavior, high performance and high accuracy, and application transparency.

Finally, we put forth the idea of running graph workloads in a dynamic environment while retaining a compressed graph. We first showed that, in

dynamic environments, the state-of-the-art clustering techniques have limitations, and result in low performance and high memory overheads. Hence, we proposed KeepCompressed (KC), a new set of algorithms that address these problems. KC constructs a compressed graph by creating vertex clusters of bounded size, and dropping edges between the clusters if necessary. Further, KC retains the graph compressed by minimally incorporating updates, while consuming acceptable amounts of memory.

5.2 Future Works

Large-scale graph processing will continue to remain an important problem for many modern computing platforms including shared memory systems. There are a number of possible directions beyond this dissertation that potentially improve the efforts of speeding-up graph processing workloads using approximate and compressed graphs:

Primitives for graph summarization techniques. Graph summarization techniques are commonly used to briefly describe the information contained in a graph [67]. For example, a social network graph includes the users as individual vertices and relationships among the users (friendships, following, etc.) as edges between the vertices. Examples of graph summarization techniques, which we focused on in this dissertation, include graph sparsification [56], k-core decomposition [73], and graph compression techniques such as V-Combiner [6] and KeepCompressed. All of these techniques require a pre-processing algorithm, to summarize (or compress) a graph into a new (most likely approximate) one. The pre-processing algorithm consists of several parallel operators on the set of vertices or edges in a graph. Generally, the behavior of each operator varies based on one or several parameter knobs. For example, sparsification is a summarization technique that randomly omits edges from a graph based on a sparsification parameter varying between 0 and 1. The higher the sparsification parameter, the less the number of omitted edges. Likewise, KeepCompressed has a mechanism to omit edges between clusters of a compressed graph given a threshold. Vertex clustering compression techniques such as V-Combiner and KeepCompressed consist of several parallel operators whose behavior can be tuned using vertex degree thresholds or maximum cluster sizes.

Designing primitives for such operators can be beneficial for several reasons. First, the graph algorithm developers save time to program their summarization technique if they decide to accelerate their graph algorithm by constructing an approximate graph. Second, using primitives facilitates the adaptive tuning of the summarization or compression parameter knobs used in different operators of the entire pre-processing algorithm. For example, these primitives can be used in a larger framework in an online graph processing environment where information about the previous executions of the primitives (on previous versions of the graph) can be used for future compression or summarization of the graph. Finally, using primitives facilitates the reasoning about the expected accuracy of the underlying graph algorithm execution. Using a similar methodology to the concept of "unit tests" from the field of software engineering, we can evaluate and reason about the effect of individual operators with different parameter knobs on the output accuracy of the graph algorithm. Hence, the use of primitives accelerates the development of effective graph compression or summarization techniques to speed-up the execution of graph processing algorithms.

Concurrent queries and graph updates. KeepCompressed is a graph compression technique suited for dynamic graphs. A key assumption in many dynamic or streaming graph processing frameworks such as KickStarter [111], GraphBolt [95], and DZiG is that updates and queries occur in a serialized fashion. In contrast, KeepCompressed potentially allows for concurrent graph queries, since it is based on Aspen [101], which allows for faster overall execution times of dynamic graph processing workloads with concurrent graph updates and queries using low-cost versioning. Other graph summarization and compression techniques studied in this dissertation such as V-Combiner, sparsification, and k-core can also benefit from concurrent queries and updates if used in dynamic graph processing settings. In addition to the versioning mechanism used by Aspen, fine-grained synchronization mechanisms on individual vertices or edges can be used to mutate the graph data-structure (ideally in its compressed form) using graph updates, while allowing for the graph queries to read the graph data-structure. The main challenge in using such technique is how to minimally incorporate the updates in the compressed graph. Overall, the ideas put forth in this dissertation can be potentially extended to address the problem of retaining compressed graphs for the case of concurrent queries and graph updates.

REFERENCES

[1] *Web Data Commons, Hyperlink graph*, Accessed October 19, 2020. [Online]. Available: http://webdatacommons.org/hyperlinkgraph/

[2] *San-Diego Supercomputer Center*, Accessed October 19, 2020. [Online]. Available: $https://www.sdsc.edu/services/hpc/hpc_systems.html$

[3] R. Nasre, M. Burtscher, and K. Pingali, "Data-driven versus topology-driven irregular computations on gpus," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE, 2013, pp. 463–474.

[4] S. Beamer, K. Asanović, and D. Patterson, "The gap benchmark suite," *arXiv preprint arXiv:1508.03619*, 2015.

[5] A. Heidarshenas, T. Gangwani, S. Yesil, A. Morrison, and J. Torrellas, "Snug: architectural support for relaxed concurrent priority queueing in chip multiprocessors," in *Proceedings of the 34th ACM International Conference on Supercomputing*, 2020, pp. 1–13.

[6] A. Heidarshenas, S. Yesil, D. Skarlatos, S. Misailovic, A. Morrison, and J. Torrellas, "V-combiner: speeding-up iterative graph processing on a shared-memory platform with vertex merging," in *Proceedings of the 34th ACM International Conference on Supercomputing*, 2020, pp. 1–13.

[7] E. W. Dijkstra, "A Note on Two Problems in Connexion with Graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, Dec. 1959.

[8] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 456–471.

[9] R. C. Prim, "Shortest connection networks and some generalizations," *Bell System Technical Journal*, vol. 36, no. 6, 1957.

[10] R. M. Fujimoto, "Parallel discrete event simulation," *Communications of ACM (CACM)*, vol. 33, no. 10, pp. 30–53, 1990. [Online]. Available: http://doi.acm.org/10.1145/84537.84545

[11] R. Bekkerman, M. Bilenko, and J. Langford, *Scaling Up Machine Learning: Parallel and Distributed Approaches.* New York, NY, USA: Cambridge University Press, 2011.

[12] I. Calciu, H. Mendes, and M. Herlihy, "The Adaptive Priority Queue with Elimination and Combining," in *Proceedings of the 28th International Symposium on Distributed Computing (DISC)*, 2014.

[13] G. C. Hunt, M. M. Michael, S. Parthasarathy, and M. L. Scott, "An Efficient Algorithm for Concurrent Priority Queue Heaps," *Information Processing Letters (IPL)*, vol. 60, no. 3, pp. 151–157, 1996.

[14] J. Lindén and B. Jonsson, "A Skiplist-Based Concurrent Priority Queue with Minimal Memory Contention," in *Proceedings of the 17th International Conference On Principles Of Distributed Systems (OPODIS)*, 2013.

[15] Y. Liu and M. Spear, "Mounds: Array-Based Concurrent Priority Queues," in *Proceedings of the 41st IEEE International Conference on Parallel Processing (ICPP)*, 2012.

[16] I. Lotan and N. Shavit, "Skiplist-Based Concurrent Priority Queues," in *Proceedings of the 14th International Parallel & Distributed Processing Symposium (IPDPS)*, 2000.

[17] H. Sundell and P. Tsigas, "Fast and lock-free concurrent priority queues for multi-thread systems," *JPDC*, vol. 65, no. 5, pp. 609–627, 2005.

[18] D. Alistarh, J. Kopinsky, J. Li, and N. Shavit, "The SprayList: A Scalable Relaxed Priority Queue," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2015.

[19] H. Rihani, P. Sanders, and R. Dementiev, "Brief Announcement: MultiQueues: Simple Relaxed Concurrent Priority Queues," in *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'15)*, 2015.

[20] M. Wimmer, J. Gruber, J. L. Träff, and P. Tsigas, "The Lock-free k-LSM Relaxed Priority Queue (Poster)," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2015.

[21] M. Wimmer, F. Versaci, J. L. Träff, D. Cederman, and P. Tsigas, "Data structures for task-based priority scheduling (poster)," in *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2014.

[22] D. R. Jefferson, "Virtual Time," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 7, no. 3, July 1985.

[23] G. Elidan, I. McGraw, and D. Koller, "Residual Belief Propagation: Informed Scheduling for Asynchronous Message Passing," in *Proceedings of the Twenty-Second Conference on Uncertainty in Artificial Intelligence (UAI)*, 2006.

[24] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and J. Shun, "Internally deterministic parallel algorithms can be fast," in *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, 2012, pp. 181–192.

[25] U. Brandes, "A faster algorithm for betweenness centrality," *Journal of Mathematical Sociology*, vol. 25, no. 2, 2001.

[26] E. Yan and Y. Ding, "Applying Centrality Measures to Impact Analysis: A Coauthorship Network Analysis," *Journal of the American Society for Information Science and Technology*, Oct. 2009.

[27] H. Yu, P. M. Kim, E. Sprecher, V. Trifonov, and M. Gerstein, "The Importance of Bottlenecks in Protein Networks: Correlation with Gene Essentiality and Expression Dynamics," *PLoS Computational Biology*, vol. 3, no. 4, Apr. 2007.

[28] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. MIT Press, 2009.

[29] K. Fraser, "Practical lock-freedom," Ph.D. dissertation, University of Cambridge, Computer Laboratory, University of Cambridge, Computer Laboratory, February 2004.

[30] W. Pugh, "Skip Lists: A Probabilistic Alternative to Balanced Trees," *CACM*, vol. 33, no. 6, pp. 668–676, June 1990.

[31] T. Harris, "A Pragmatic Implementation of Non-blocking Linked-lists," in *Proceedings of the 15th International Symposium on Distributed Computing (DISC)*, 2001.

[32] M. M. Michael, "High performance dynamic lock-free hash tables and list-based sets," in *Proceedings of the 14th ACM Symposium on Parallel Algorithms and Architectures (SPAA '02)*, 2002.

[33] D. P. Bertsekas, F. Guerriero, and R. Musmanno, "Parallel Asynchronous Label-correcting Methods for Shortest Paths," *Journal of Optimization Theory and Applications*, vol. 88, no. 2, Feb. 1996.

[34] A. Lenharth, D. Nguyen, and K. Pingali, "Priority queues are not good concurrent priority schedulers," in *Proceedings of the 21st International Conference on Parallel and Distributed Computing (Euro-Par)*, 2015.

[35] NVIDIA, "CUDA Programming Guide v3.1," http://docs.nvidia.com/cuda/cuda-c-programming-guide/, 2019.

[36] Khronos Group, "OpenCL," http://www.khronos.org/opencl/, 2019.

[37] J. Matai, D. Richmond, D. Lee, Z. Blair, Q. Wu, A. Abazari, and R. Kastner, "Resolve: Generation of high-performance sorting architectures from high-level synthesis," in *Proceedings of the 24th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2016, pp. 195–204.

[38] Wikipedia, "Moving Average," https://en.wikipedia.org/wiki/Moving average, 2019.

[39] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, Aug. 2011.

[40] M. Wimmer, J. Gruber, J. Larsson Träff, and P. Tsigas, "The Lock-free k-LSM Relaxed Priority Queue," *ArXiv e-prints*, Mar. 2015.

[41] J. Karnon, J. Stahl, A. Brennan, J. J. Caro, J. Mar, and J. Möller, "Modeling using discrete event simulation: A report of the ispor-smdm modeling good research practices task force–4," *Medical decision making*, vol. 32, no. 5, pp. 701–711, 2012.

[42] M. Holly and C. Tropper, "Parallel discrete event n-body dynamics," in *Proceedings of the 2011 IEEE Workshop on Principles of Advanced and Distributed Simulation*, ser. PADS '11, 2011, pp. 1–10.

[43] DIMACS, "9th DIMACS implementation challenge," 2010, http://www.dis.uniroma1.it/challenge9/download.shtml.

[44] SNAP, "SNAP Datasets," https://snap.stanford.edu/data/amazon0302.html, 2003.

[45] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-mat: A recursive model for graph mining," in *Proceedings of the 2004 SIAM International Conference on Data Mining*. SIAM, 2004.

[46] N. Sturtevant, "Benchmarks for grid-based pathfinding," *Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 2, pp. 144 – 148, 2012. [Online]. Available: http://web.cs.du.edu/ sturtevant/papers/benchmarks.pdf

[47] M. C. Jeffrey, S. Subramanian, C. Yan, J. Emer, and D. Sanchez, "A Scalable Architecture for Ordered Parallelism," in *48th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015.

[48] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi, "CACTI 5.1," HP Labs, Tech. Rep. HPL-2008-20, 2008.

[49] J. Haj-Yihia, A. Yasin, Y. B. Asher, and A. Mendelson, "Fine-grain power breakdown of modern out-of-order cores and its implications on skylake-based systems," *ACM Trans. Archit. Code Optim.*, vol. 13, no. 4, pp. 56:1–56:25, Dec. 2016. [Online]. Available: http://doi.acm.org/10.1145/3018112

[50] U. Kang, D. Horng et al., "Inference of beliefs on billion-scale graphs," *Workshop on Large-scale Data Mining: Theory and Applications*, 2010.

[51] U. Kang, D. H. Chau, and C. Faloutsos, "Mining large graphs: Algorithms, inference, and discoveries," in *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 2011, pp. 243–254.

[52] X. Zhu and Z. Ghahramani, "Learning from labeled and unlabeled data with label propagation," Carnegie Mellon University, Tech. Rep., 2002.

[53] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Tech. Rep., 1999.

[54] J. M. Kleinberg, "Authoritative sources in a hyperlinked environment," *Journal of the ACM (JACM)*, vol. 46, no. 5, pp. 604–632, 1999.

[55] A. Kusum, K. Vora, R. Gupta, and I. Neamtiu, "Efficient processing of large graphs via input reduction," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2016, pp. 245–257.

[56] A. P. Iyer, A. Panda, S. Venkataraman, M. Chowdhury, A. Akella, S. Shenker, and I. Stoica, "Bridging the gap: towards approximate graph analytics," in *Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*. ACM, 2018, p. 10.

[57] I. Mitliagkas, M. Borokhovich, A. G. Dimakis, and C. Caramanis, "Frogwild!: Fast pagerank approximations on graph engines," *Proceedings of the VLDB Endowment*, vol. 8, no. 8, pp. 874–885, 2015.

[58] Y. Fujiwara, M. Nakatsuji, H. Shiokawa, T. Mishima, and M. Onizuka, "Fast and exact top-k algorithm for pagerank," in *Twenty-Seventh AAAI Conference on Artificial Intelligence*, 2013.

[59] B. Xiang, Q. Liu, E. Chen, H. Xiong, Y. Zheng, and Y. Yang, "Pagerank with priors: An influence propagation perspective," in *Twenty-Third International Joint Conference on Artificial Intelligence*, 2013.

[60] K. Tretyakov, A. Armas-Cervantes, L. García-Bañuelos, J. Vilo, and M. Dumas, "Fast fully dynamic landmark-based estimation of shortest path distances in very large graphs," in *Proceedings of the 20th ACM international conference on Information and knowledge management*, 2011, pp. 1785–1794.

[61] A. P. Iyer, Z. Liu, X. Jin, S. Venkataraman, V. Braverman, and I. Stoica, "Asap: Fast, approximate graph pattern mining at scale," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 745–761.

[62] I. Goiri, R. Bianchini, S. Nagarakatte, and T. D. Nguyen, "ApproxHadoop: Bringing Approximations to MapReduce Frameworks," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2015, pp. 383–397.

[63] Z. Shang and J. X. Yu, "Auto-approximation of graph computing," *Proceedings of the VLDB Endowment*, vol. 7, no. 14, pp. 1833–1844, 2014.

[64] H. Omar, M. Ahmad, and O. Khan, "Graphtuner: An input dependence aware loop perforation scheme for efficient execution of approximated graph algorithms," in *2017 IEEE International Conference on Computer Design (ICCD)*. IEEE, 2017, pp. 201–208.

[65] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard, "Managing performance vs. accuracy trade-offs with loop perforation," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 124–134.

[66] M. Rinard, "Probabilistic accuracy bounds for fault-tolerant computations that discard tasks," in *Proceedings of the 20th annual international conference on Supercomputing*. ACM, 2006, pp. 324–334.

[67] Y. Liu, T. Safavi, A. Dighe, and D. Koutra, "Graph summarization methods and applications: A survey," *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, p. 62, 2018.

[68] T. Sarlós, A. A. Benczúr, K. Csalogány, D. Fogaras, and B. Rácz, "To randomize or not to randomize: space optimal summaries for hyperlink analysis," in *Proceedings of the 15th international conference on World Wide Web*. ACM, 2006, pp. 297–306.

[69] T. Akiba and Y. Yano, "Compact and scalable graph neighborhood sketching," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2016, pp. 685–694.

[70] D. Dor, S. Halperin, and U. Zwick, "All-pairs almost shortest paths," *SIAM Journal on Computing*, vol. 29, no. 5, pp. 1740–1759, 2000.

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

[72] J. Batson, D. A. Spielman, N. Srivastava, and S.-H. Teng, "Spectral sparsification of graphs: theory and algorithms," *Communications of the ACM*, vol. 56, no. 8, pp. 87–94, 2013.

[73] W. Khaouid, M. Barsky, V. Srinivasan, and A. Thomo, "K-core decomposition of large networks on a single pc," *Proceedings of the VLDB Endowment*, vol. 9, no. 1, pp. 13–23, 2015.

[74] A. Pinar, T. G. Kolda, and C. Peng, "Accelerating community detection by using k-core subgraphs." Sandia National Lab.(SNL-CA), Livermore, CA (United States), Tech. Rep., 2014.

[75] P. Govindan, C. Wang, C. Xu, H. Duan, and S. Soundarajan, "The k-peak decomposition: Mapping the global structure of graphs," in *Proceedings of the 26th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 2017, pp. 1441–1450.

[76] A. Kyrola, G. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a PC," in *Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012, pp. 31–46.

[77] Y. Kozawa, T. Amagasa, and H. Kitagawa, "Gpu-accelerated graph clustering via parallel label propagation," in *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*. ACM, 2017, pp. 567–576.

[78] J. Ugander and L. Backstrom, "Balanced label propagation for partitioning massive graphs," in *Proceedings of the sixth ACM international conference on Web search and data mining*. ACM, 2013, pp. 507–516.

[79] H. Bao and E. Y. Chang, "AdHeat: An influence-based diffusion model for propagating hints to match ads," in *Proceedings of the 19th international conference on World wide web*. ACM, 2010, pp. 71–80.

[80] J. Malicevic, B. Lepers, and W. Zwaenepoel, "Everything you always wanted to know about multicore graph processing but were afraid to ask," in *2017 USENIX Annual Technical Conference (USENIXATC 17)*, 2017, pp. 631–643.

[81] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, June 2014.

[82] R. Meusel, O. Lehmberg, C. Bizer, and S. Vigna, "Web data commons - hyperlink graphs," http://webdatacommons.org/hyperlinkgraph/, 2019.

[83] D. R. Karger and C. Stein, "A new approach to the minimum cut problem," *Journal of the ACM (JACM)*, vol. 43, no. 4, pp. 601–640, 1996.

[84] J. Kunegis, "Konect: the koblenz network collection," in *Proceedings of the 22nd International Conference on World Wide Web*. ACM, 2013, pp. 1343–1350.

[85] L. Dhulipala, G. Blelloch, and J. Shun, "Julienne: A framework for parallel graph algorithms using work-efficient bucketing," in *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, 2017, pp. 293–304.

[86] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, pp. 1–25, 2011.

[87] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2012, pp. 449–460.

[88] W. Gatterbauer, "The linearization of belief propagation on pairwise markov random fields," in *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[89] J. Yoo, S. Jo, and U. Kang, "Supervised belief propagation: Scalable supervised inference on attributed networks," in *2017 IEEE International Conference on Data Mining (ICDM)*. IEEE, 2017, pp. 595–604.

[90] M.-H. Jang, C. Faloutsos, S.-W. Kim, U. Kang, and J. Ha, "Pin-trust: Fast trust propagation exploiting positive, implicit, and negative information," in *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. ACM, 2016, pp. 629–638.

[91] K. Jung, W. Heo, and W. Chen, "Irie: Scalable and robust influence maximization in social networks," in *2012 IEEE 12th International Conference on Data Mining*. IEEE, 2012, pp. 918–923.

[92] A. Fidel, F. C. Sabido, C. Riedel, N. M. Amato, and L. Rauchwerger, "Fast approximate distance queries in unweighted graphs using bounded asynchrony," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2016, pp. 40–54.

[93] X. Sui, A. Lenharth, D. S. Fussell, and K. Pingali, "Proactive control of approximate programs," *ACM SIGOPS Operating Systems Review*, vol. 50, no. 2, pp. 607–621, 2016.

[94] A. Grewal, J. Jiang, G. Lam, T. Jung, L. Vuddemarri, Q. Li, A. Landge, and J. Lin, "Recservice: Distributed real-time graph processing at twitter," in *10th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 18)*, 2018.

[95] M. Mariappan and K. Vora, "Graphbolt: Dependency-driven synchronous processing of streaming graphs," in *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019, pp. 1–16.

[96] M. Mariappan, J. Che, and K. Vora, "Dzig: sparsity-aware incremental processing of streaming graphs," in *Proceedings of the Sixteenth European Conference on Computer Systems*, 2021, pp. 83–98.

[97] K. Shin, A. Ghoting, M. Kim, and H. Raghavan, "Sweg: Lossless and lossy summarization of web-scale graphs," in *The World Wide Web Conference*, 2019, pp. 1679–1690.

[98] S. Navlakha, R. Rastogi, and N. Shrivastava, "Graph summarization with bounded error," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, 2008, pp. 419–432.

[99] K. A. Kumar and P. Efstathopoulos, "Utility-driven graph summarization," *Proceedings of the VLDB Endowment*, vol. 12, no. 4, pp. 335–347, 2018.

[100] M. Riondato, D. García-Soriano, and F. Bonchi, "Graph summarization with quality guarantees," *Data mining and knowledge discovery*, vol. 31, no. 2, pp. 314–349, 2017.

[101] L. Dhulipala, G. E. Blelloch, and J. Shun, "Low-latency graph streaming using compressed purely-functional trees," in *Proceedings of the 40th ACM SIGPLAN conference on programming language design and implementation*, 2019, pp. 918–934.

[102] D. Ediger, R. McColl, J. Riedy, and D. A. Bader, "Stinger: High performance data structure for streaming graphs," in *2012 IEEE Conference on High Performance Extreme Computing*. IEEE, 2012, pp. 1–5.

[103] U. A. Acar, D. Anderson, G. E. Blelloch, and L. Dhulipala, "Parallel batch-dynamic graph connectivity," in *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, 2019, pp. 381–392.

[104] L. Dhulipala, D. Durfee, J. Kulkarni, R. Peng, S. Sawlani, and X. Sun, "Parallel batch-dynamic graphs: Algorithms and lower bounds," in *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 2020, pp. 1300–1319.

[105] N. Simsiri, K. Tangwongsan, S. Tirthapura, and K.-L. Wu, "Work-efficient parallel union-find with applications to incremental graph connectivity," in *European Conference on Parallel Processing*. Springer, 2016, pp. 561–573.

[106] B. Wheatman and H. Xu, "Packed compressed sparse row: A dynamic graph representation," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 2018, pp. 1–7.

[107] P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer, "Llama: Efficient graph analytics using large multiversioned arrays," in *2015 IEEE 31st International Conference on Data Engineering*. IEEE, 2015, pp. 363–374.

[108] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher, "Min-wise independent permutations," *Journal of Computer and System Sciences*, vol. 60, no. 3, pp. 630–659, 2000.

[109] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *AAAI*, 2015. [Online]. Available: http://networkrepository.com

[110] J. Leskovec and C. Faloutsos, "Sampling from large graphs," in *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2006, pp. 631–636.

[111] K. Vora, R. Gupta, and G. Xu, "Kickstarter: Fast and accurate computations on streaming graphs via trimmed approximations," *ACM SIGOPS Operating Systems Review*, vol. 51, no. 2, pp. 237–251, 2017.