Understanding Priority-Based Scheduling of Graph Algorithms on a Shared-Memory Platform

Serif Yesil
University of Illinois at Urbana-Champaign
syesil2@illinois.edu

Azin Heidarshenas
University of Illinois at Urbana-Champaign
heidars2@illinois.edu

Adam Morrison
Tel Aviv University
mad@cs.tau.ac.il

Josep Torrellas
University of Illinois at Urbana-Champaign
torrella@illinois.edu

ABSTRACT

Many task-based graph algorithms benefit from executing tasks according to some programmer-specified priority order. To support such algorithms, graph frameworks use Concurrent Priority Schedulers (CPSs), which attempt—but do not guarantee—to execute the tasks according to their priority order. While CPSs are critical to performance, there is insufficient insight on the relative strengths and weaknesses of the different CPS designs in the literature. Such insights would be valuable to design better CPSs for graph processing.

This paper addresses this problem. It performs a detailed empirical performance analysis of several advanced CPS designs in a state-of-the-art graph analytics framework running on a large shared-memory server. Our analysis finds that all CPS designs but one impose major overheads that dominate running time. Only one CPS—the Galois system's `obim`—typically imposes negligible overheads. However, `obim`'s performance is input-dependent and can degrade substantially for some inputs. Based on our insights, we develop *PMOD*, a new CPS that is robust and delivers the highest performance overall.

CCS CONCEPTS

• **Computing methodologies → Shared memory algorithms**.

KEYWORDS

Concurrent Priority Scheduling, Graphs, Shared-Memory Multiprocessors, Memory Hierarchy.

ACM Reference Format:
Serif Yesil, Azin Heidarshenas, Adam Morrison, and Josep Torrellas. 2019. Understanding Priority-Based Scheduling of Graph Algorithms on a Shared-Memory Platform. In *The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '19), November 17–22, 2019, Denver, CO, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3295500.3356160

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SC '19, November 17–22, 2019, Denver, CO, USA

© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-6229-0/19/11…$15.00
https://doi.org/10.1145/3295500.3356160

1 INTRODUCTION

The fundamental role that graph algorithms play in many important applications motivates the use of parallelism to speed them up. As a result, there is a large body of work on programming models and runtimes for parallel graph processing (e.g., [9, 21, 26, 31, 32, 37]). Many of these frameworks use a task-based model on a shared-memory environment. In this model, the graph algorithm's computation is broken down into dynamically-created tasks that are scheduled to run in parallel. This is an attractive model, as it is very general, reasonably easy to program, and can be executed efficiently on large commercial shared-memory machines [26].

Task-based graph algorithms are usually unordered. This means that tasks can be processed in any order. However, many unordered algorithms benefit from executing tasks according to some programmer specified priority order. For instance, consider the single-source shortest paths (SSSP) problem, which computes the shortest distance from a source vertex s to every vertex in the graph. It is more efficient to process vertices roughly ordered in increasing distance from s. If distant vertices are processed first, the execution will likely discover shorter paths to those vertices later, making the earlier computation on the distant vertices redundant.

Graph algorithms that benefit from task processing in priority order are ubiquitous. They include search algorithms, such as SSSP and Breadth-First Search (BFS), and path-finding algorithms, such as A*, which are used for gaming, transportation, and robotics [23]. They also include PageRank [34], which is widely used for graph analytics, Delaunay triangulation [3] for computational geometry, maximal flow computation [8] for optimization and scheduling, and minimal spanning tree (MST) finding [6] for network design.

To run such algorithms efficiently, graph frameworks use a *Concurrent Priority Scheduler* (CPS) data structure to pick tasks for execution roughly in their priority order [2, 18, 26, 29, 35, 36]. While CPSs are critical to the performance of these graph applications, there is insufficient insight on the relative strengths and weaknesses that the CPS designs in the literature exhibit in practice. In particular, many of the designs have never been evaluated against each other in a state-of-the-art graph analytics system on a large shared-memory machine. Further, there is no detailed low-level empirical analysis of the sources of CPS overhead. Only with this type of empirical analysis can one identify the bottlenecks and get insights into how to design better CPSs for graph processing.

This paper addresses this open question. It performs a detailed empirical performance analysis of several advanced CPS designs using Galois [26], a state-of-the-art graph analytics framework,

running on a 40-core, 4-socket, shared-memory machine. We study four state-of-the-art CPS designs that use different, representative structures: a SprayList [2] using a shared lock-free skip list, two variations of a distributed array of priority queues [29], and `obim`, the default CPS in Galois.

All the CPSs we study avoid synchronization bottlenecks by having processors pick *some* high-priority task rather than the highest-priority one. Despite this property, we find that all CPSs but `obim` impose significant overhead on the fine-grained tasks of graph algorithms, typically causing CPS overhead to dominate execution time. We further observe that `obim`'s performance depends on having many tasks per each priority value. When there are few tasks per priority level, `obim`'s execution becomes slower than sequential execution—even if the input creates abundant task parallelism. One currently has to *manually* tune the graph application to work around `obim`'s performance stability problem.

Guided by our study, we develop *PMOD*, a new CPS that extends `obim` to dynamically and automatically adapt to the ranges of priorities exhibited by the input graph. This property makes PMOD's performance stable, automatically achieving comparable performance to `obim` with input-specific manual tuning. PMOD obtains the highest performance of all CPSs.

Contributions. We make the following contributions.

- We conduct the first extensive empirical analysis of different CPS algorithms proposed in the literature. Our analysis on a large shared-memory machine yields qualitative and quantitative insights about the trade-offs in CPS design, to drive future research on CPSs and graph applications.
- Our analysis points out a missing point in the CPS design space: a CPS that provides high performance in a consistent manner that does not depend on the input, and that does not require manual tuning. We propose the new *PMOD* CPS, which has these properties.

All of our software infrastructure is available at https://github.com/serifyesil/PMOD.

2 BACKGROUND AND MOTIVATION

This paper focuses on graph analytics applications running on a shared-memory machine. The graph, as well as data associated with vertices and edges, is stored in a standard graph representation, such as compressed sparse row (CSR) or column (CSC). The graph representation does not affect our discussion or findings. The graph applications can read and write the data associated with vertices and edges, and update the graph structure by adding/removing vertices and edges.

2.1 Priority Scheduling

Graph analytics systems employ different programming models for parallel graph algorithms [9, 21, 26, 31, 32, 37]. We focus on the popular *task-based* model, in which the algorithm is implemented using dynamically-created tasks that may run in parallel. The task-based programming model can express algorithms designed for other models, and can run very efficiently on current machines. For example, Galois [26]—one of the best performing shared-memory graph analytics systems—uses this model.

In a task-based graph algorithm, each task performs vertex and/or edge updates, and can also create new tasks. Algorithms can

```
1  Shared data: Graph G. Initially , in each vertex v, v.dist = ∞
```

(a) ordered:

```
2  sssp(s):
3    schedule( visit (s, 0))
4
5  visit(v, dist):
6    if v.dist ≠ ∞: return
7    v.dist = dist
8    foreach neighbor u of v:
9      if (u.dist == ∞):
10       d = dist + weight(v, u)
11       schedule( visit (u, d))
12
```

(b) unordered:

```
13 sssp(s):
14   s.dist = 0
15   schedule( visit (s, 0))
16 visit(v, dist):
17   if v.dist ≠ dist: return
18   foreach neighbor u of v:
19     atomically :
20       d = dist + weight(v, u)
21       if (d < u.dist)
22         u.dist = d
23         schedule( visit (u, d))
```

(a) ordered **(b) unordered**

Figure 1: Task-based SSSP algorithm. The priority of a task is *dist*.

be ordered or unordered. An *ordered* algorithm *requires* tasks to execute according to a user-specified priority order. Figure 1a shows an example ordered algorithm implementing Dijkstra's single-source shortest paths (SSSP) algorithm [10]. The algorithm finds the distance from some source vertex s to all other vertices in a weighted directed graph G. Each task processes a vertex, and its priority is the length of the path it discovers from the source to that vertex. A task attempts to extend the shortest path by creating a task for each neighbor. Tasks must run in strict priority order.

Ordered algorithms have parallelism [13, 15], but mining it is hard, since it requires speculating across ordering constraints. Efficiently performing such speculation requires special hardware (e.g., [15]), because the overheads of software-based speculation negate the parallelism benefits [13]. Consequently, graph analytics systems favor unordered algorithms.

An *unordered* algorithm produces a correct result regardless of task execution order, making it easier to mine its parallelism. Figure 1b shows an unordered SSSP algorithm. Tasks now do not necessarily terminate if the vertex has already been visited. In addition, they update the distance only if they decrease it, which requires synchronization (Lines 19–23).

Many unordered algorithms still use priorities, running more efficiently when task execution mostly follows priority order, but remaining correct when tasks execute out of priority order (i.e., *priority inversion*). For instance, executing an SSSP task out of order can lead to the task's distance update being overwritten later, thereby wasting the task's cycles. Therefore, graph analytics systems use *priority scheduling*, which attempts—but does not *guarantee*—to execute tasks in priority order.

2.2 Concurrent Priority Schedulers

A *Concurrent Priority Scheduler* (CPS) is a data structure that stores the set of pending tasks, and provides a way to add and remove tasks. A CPS supports two main operations: `Enq` and `Deq`. An `Enq`(t, p) operation enqueues a task t with priority p in the data structure. A `Deq`() operation dequeues a task to execute. The execution consists of cores repeatedly invoking `Deq` and executing the obtained task (which invokes `Enq` if it creates new tasks) until no tasks are left.

A CPS can be implemented by a concurrent Priority Queue (PQ) [20, 30, 33], in which a `Deq` returns the highest priority task (i.e., the one with minimal p value). In this case, all concurrent `Deq` calls contend on the same task, inducing synchronization overhead. Therefore, practical CPS designs *relax* the priority queue's semantics and return *some* high-priority task, not necessarily the highest-priority one.

We consider several representative state-of-the-art CPS designs:

SprayList. The SprayList [2] is a popular design that stores tasks in priority order inside a lock-free skip list [12].[1] A SprayList `Enq` inserts the task into the skip list, which is sorted by priority order. Lock-free skip list insertions are not serialized and run concurrently. A SprayList `Deq` operation removes a random high priority task, which it finds by performing a short random walk on the skip list. Different processors thus typically pick different tasks and do not contend. A `Deq` returns one of the $\approx p \log^3 p$ highest-priority tasks with high probability, where p is the number of processors.

Distributed Queues. These designs reduce contention by maintaining an array of concurrent PQs, and allowing a processor to access a random PQ in each operation. Processors thus typically access different PQs and do not contend.

We consider two designs that differ in how operations access the PQs: MultiQueue and RELD. The MultiQueue [29] maintains an array of $q = cp$ concurrent PQs, where $c > 1$ is a parameter, and p is the number of processors. An `Enq` inserts the task into a random PQ. A `Deq` picks two random PQs and removes the task of higher priority among the two. Recent work suggests that, in expectation, the MultiQueue `Deq` picks one of the $\approx p$ globally highest-priority tasks (i.e., over all PQs) [1].

`RELD` (random enqueue, local dequeue) maintains an array of p concurrent PQs, each of which is associated with a processor. As in the MultiQueue, an `Enq` inserts the task into a random PQ. A `Deq` dequeues from the requesting processor's PQ, blocking if it is empty. A hardware implementation of `RELD` is used by the Swarm architecture [15].

Galois obim. Galois' default CPS is `obim` (Ordered By Integer Metric) [26, 27], which strives to avoid communication and synchronization between processors. This is a lightweight, distributed design, with one *bag* (i.e., unordered queue) data structure per priority. Each bag is a distributed structure consisting of as many FIFO queues as sockets (i.e., NUMA domains) in the machine.

An `Enq` inserts a task into the bag associated with the task's priority, creating such a bag if it does not exist. A `Deq` finds a bag to dequeue from by traversing the bags in priority order until it finds a non-empty bag. The processor keeps dequeueing from this bag in subsequent `Deq`s until it becomes empty.

The bag is designed to minimize communication and synchronization by satisfying most `Enq` and `Deq` operations from private per-processor buffers, so that processors access the shared FIFO queues only infrequently. A bag's FIFO queues hold *chunks* of c tasks (typically, $c = 64$). When a processor inserts tasks into the bag, it first buffers them in a private local chunk; once the chunk fills up, the processor enqueues it into the FIFO queue of the processor's socket. The tasks in the chunk then become visible to other processors. A processor dequeues a chunk from its socket's queue. If the queue is empty, the processor steals from one of the remote queues. It then consumes tasks from the dequeued chunk one at a time.

`obim` maintains the list of bags in a *global map* data structure, which is read and written by all threads. To reduce synchronization and cache coherence traffic, each thread caches the contents of the global map in a *local map*. When enqueueing a task, a thread looks up the bag associated with the task's priority in its local map. If not found, the thread creates a new bag and updates the global map accordingly. When dequeueing, if a thread fails to find work in the bags listed in its local map, it refreshes the local map with the information in the global map, and tries again.

[1] A skip list [28] is a randomized list-based data structure in which nodes are randomly linked into a hierarchy of linked lists. With high probability, each list contains about half of the nodes in the list below it, allowing searches to "skip" over multiple elements.

3 PRIORITY SCHEDULING INSIGHTS

3.1 Fundamental Tradeoff

The fundamental tradeoff in CPS design is that of communication and synchronization overhead versus unnecessary work performed. Specifically, if the CPS is such that tasks are obtained and processed in perfect priority order, the algorithm typically performs the least amount of work. However, the communication and synchronization operations necessary to obtain the tasks in such order are costly.

Instead, if the CPS obtains tasks without following strict priority order, there is a chance that some of the work performed will be superfluous; it will have to be repeated under more up-to-date conditions. However, by relaxing priority order, the CPS can reduce communication and synchronization.

We classify the execution cycles of an unordered graph algorithm that uses a CPS as shown in Table 1. The algorithm's cycles spent processing tasks can be performing Good Work (*GWork*) or Useless Work (*UWork*). The algorithm's cycles spent in the CPS can be Enqueue (*Enq*), Dequeue (*Deq*), and Failed Dequeue (*FDeq*) cycles. The latter occur when a dequeue fails to find a task to execute. The remainder Other cycles in the algorithm are spent running other framework code. Typically, if strict priority execution is maintained, *Enq* and *Deq* will be high, but *UWork* will be low. If priority execution is relaxed, the opposite will occur. Our goal is to find a balance for the best performance.

Table 1: Execution cycle breakdown of unordered graph algorithms.

Category	Description
Good Work (GWork)	Processing a task. The work ends up being useful.
Useless Work (UWork)	Processing a task. The work later proves useless.
Enqueue (Enq)	Pushing a task to the CPS data structure.
Dequeue (Deq)	Retrieving a task from the CPS data structure.
Failed Dequeue (FDeq)	Attempting and failing to retrieve a task from the CPS data structure.
Other	Executing other graph analytics framework code.

3.2 Addressing the Tradeoff

We posit that there are two main approaches to address the outlined tradeoff: one that emphasizes reduction in useless work and another that emphasizes reduction in communication/synchronization overhead.

3.2.1 *Emphasis on Minimizing Useless Work:*

CPSs that emphasize retrieving tasks close to the priority order invest synchronization/-communication operations to obtain high-quality tasks. Although

modern CPSs avoid the contention bottleneck of dequeueing tasks from a single shared priority queue [28, 30], they still access globally shared data on each CPS operation, which typically incurs multiple cache misses.

Specifically, the SprayList maintains a global skip list, from which it dequeues a random task close to the head. In the distributed queue designs such as the MultiQueue and RELD, threads access a random queue in most operations. The MultiQueue picks a random queue to enqueue and dequeue, while RELD enqueues in a random remote queue and dequeues locally. The use of randomness in these CPSs causes every CPS operation that a thread performs to access (with high probability) different memory locations than those accessed by its previous CPS operation. These locations are also frequently written to (e.g., queue heads in the MultiQueue and RELD). Consequently, despite returning high-quality tasks, CPS operations in these designs incur multiple cache misses, and are thus relatively time consuming compared to the fine-grained tasks used in graph applications.

3.2.2 Emphasis on Minimizing Communication: The `obim` CPS [26] exemplifies a CPS design that prioritizes avoiding shared-memory communication, maximizing locality of CPS operations, and minimizing their overhead. It maintains tasks in *per-priority distributed* unordered queues (bags). CPS operations on these bags are efficient and highly local. First, tasks can be inserted/removed at the per-priority queue tail/head without any list traversal, as their order is not important. Second, to amortize overheads, enqueues and dequeues are performed at a coarse grain, by enqueueing and dequeueing a chunk of tasks at a time [26]. Such amortization is not trivial to add to the first approach to CPS designs.

For the `obim` design to be efficient, a worker thread must have a fast way to find the bag associated with a priority value. Further, the per-priority queues should contain many tasks.

In principle, this design is prone to useless work, because threads working on a bag do not frequently search for a new bag that could have a higher priority, to reduce communication. We shall see, however, that such useless work is typically rare in practice.

We call this CPS approach *Per-Priority Queue*, and the first approach, which includes the SprayList, MultiQueue, and RELD CPSs, *Combined-Priority Queue*.

3.3 Observations

We analyzed the execution of several graph algorithms on a large multi-socket shared-memory server with the CPS implementations described in Section 2.2. We ran the algorithms on many different graph inputs and various thread counts. Our main observations are shown in Table 2. Detailed measurements supporting these observations are presented in Section 6.

In *O1*, we observe that some task processing with out-of-order priorities (i.e., *priority inversion*) can be a good choice if done on communication-minimizing CPS implementations such as those of Section 3.2.2. These CPSs have very low-overhead operations, while producing acceptable amounts of useless work. Even with the useless work, the overall result is higher performance than other CPSs, especially for large core counts.

The main reason why *O1* holds is shown in *O2*: when a task with priority p is processed, it often tends to generate other tasks with only slightly lower priorities ($p+\epsilon$, where ϵ is small[2]). This property means that if threads are working on the highest priority bags, newly created tasks do not change the highest priority, and so continuing to work on the bag does not create useless work. Moreover, even if we place these new tasks into the current bag (as discussed in Section 4), the bag will contain tasks with similar priorities, and so processing the new tasks will cause only minor priority inversion.

Table 2: Observations on successful CPS designs.

Observation	Description
O1	Some task processing with priority inversion is a good choice, if it is the result of a lightweight CPS.
O2	The number of tasks per priority is input-dependent. Typically, processing a task T generates new tasks with priorities not too different from T's priority. However, with some graph inputs, these new tasks have a very wide range of priorities, with negative performance effects for the Per-Priority Queue approach.
O3	Enqueueing and dequeueing a chunk of tasks at a time is very beneficial, but does not seem compatible with the Combined-Priority Queue approach.

The properties of input graphs cause this behavior. Processing a vertex *v* may cause the insertion of its adjacent vertex *v'* in the work queue. But the priority of *v'* is often not much different than that of *v*. For example, in SSSP, the difference is the weight of the edge joining *v* to *v'*; in BFS, the difference is 1. As another example, in algorithms like MST, the priority is simply the degree (i.e., the number of edges) of a vertex. Such numbers are often not very different.

Observation *O2* also notes that, sometimes, applications or inputs generate tasks with a wide range of priorities, yielding a small number of tasks per priority value. This causes poor performance in CPSs using the Per-Priority Queue approach. If there are only a few tasks per priority, the private chunks where threads buffer created tasks typically fail to fill up, and thus the tasks they contain never become visible to other threads. Consequently, many bags will appear empty, while all the tasks are stored in threads' private chunks. This situation causes the algorithm to spend a lot of time searching for bags to work on. Moreover, threads will quickly empty any bags found and thus not benefit from locality.

On the other hand, the Combined-Priority Queue approach is more tolerant of this behavior. This is because it orders the tasks according to their priority in a queue. The queue is processed in the same way, irrespective of the ranges of priorities it contains.

This effect happens in SSSP with many *road network* graphs. The difference in priorities between two adjacent vertices is the weight of the connecting edge. This is the distance between the two corresponding vertices. A vertex may be connected to several vertices at widely different distances. As a result, the range of priorities can be very large, causing major dequeueing overheads with the Per-Priority Queue approach.

Finally, *O3* notes that a lot of execution overhead is eliminated by performing enqueueing and dequeueing of tasks in a coarse-grain manner—i.e., using chunks of tasks at a time. Such approach is easy to support with the Per-Priority Queue approach. However,

[2] Higher p values mean lower priorities.

it is hard to support with the Combined-Priority Queue approach without destroying its robustness to the priority distribution (see *O2*). It is not possible to simply create chunks based on inserted tasks, because the resulting chunks would contain tasks with different priorities, and so these chunks would not be totally ordered in the queue. Alternatively, it is possible for each thread to buffer the tasks it creates in per-priority private chunks and insert filled-up chunks into the global queue (similarly to the Per-Priority Queue approach). However, this design suffers from the problem noted in *O2*—when there are few tasks per priority, these chunks will not fill up and will not be inserted into the queue.

We also find that large chunks are undesirable, as they lead to load imbalance among cores. Indeed, in a chunk-based environment, a core bundles up the work that it is generating in chunks, before enqueuing the chunks in the work list. Only at that point is the work visible to other cores. If chunks are large, it takes a long time for a core to fill up a chunk and make it globally visible. During that time, other cores may be idle looking for work.

4 PMOD: AN ADAPTIVE CPS

4.1 Main Idea

Based on our observations, we introduce a new CPS design that is able to minimize both communication/synchronization overhead and unnecessary work performed, hence delivering high performance. Our scheme is called PMOD (Priority Merging On Demand), and builds on the ideas in Section 3.2.2, which `obim` implements.

While `obim`'s idea of keeping a queue per priority is often highly effective, it can sometimes result in subpar performance. Hence, in PMOD, the queues are *per priority groups*, and such groups change dynamically at runtime.

Specifically, PMOD dynamically identifies when the execution is using too many priority queues and there are too few tasks per priority queue. This is an inefficient operating point because threads spend substantial time searching for work. In this case, PMOD combines a set of consecutive priorities into a single queue. We call this process *Priority Merging*. This process can be repeated multiple times dynamically. Every time, the *Merging Factor* (or number of consecutive priorities that are merged) increases. The Merging Factor is always a power of two.

PMOD also dynamically estimates when the execution is using too few priority queues. This is also an inefficient operation point because, by merging disparate priorities, threads run the risk of suffering priority inversion and executing useless work. In this case, PMOD separates the priorities into more queues. This is *Priority Unmerging*. It is done dynamically, in decreasing powers of two.

To see how the algorithm works, consider Figure 2. Figure 2a shows an environment with too many priority queues. Core i has a long list of priority queues in its local map (Section 2.2), but they are all currently empty, because no chunk has been filled-up and deposited in any of these queues. Core i wastes time traversing this list, and then has to go to the global map to obtain work. Our PMOD CPS measures the frequency of such global accesses. If the frequency is higher than threshold $Freq_{global}$, PMOD considers priority merging.

In this case, PMOD first computes the range of priorities of the tasks that have been recently enqueued in the work queue, and

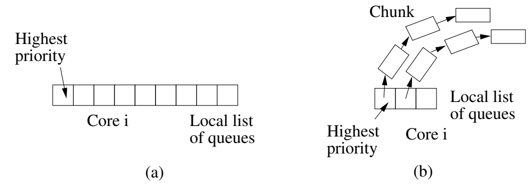


Figure 2: Run with too many (a) and too few (b) priority queues.

divides it by the Merging Factor. This gives the number of queues needed to cover the range (Num_{queues}). Then, PMOD computes the ratio between the number of tasks recently enqueued in the work queue and Num_{queues}. To reduce traversal overheads, we want this ratio to be equal or higher than threshold *MinDensity*. If necessary, PMOD increases the Merging Factor (and hence consolidates the list of queues) so that this is the case. From now on, newly arriving chunks will be enqueued in the consolidated list of queues.

Figure 2b shows an environment with too few priority queues and too many tasks per queue. It is possible that core i performs substantial useless work. Our PMOD CPS regularly measures the number of recent successful dequeues from each priority queue. If one such value is higher than threshold *MaxPops*, PMOD suspects that queues are too long and priority inversion may be taking place, and considers priority unmerging. Again, it computes the number of queues needed to cover the recent priority range (Num_{queues}). To minimize the amount of useless work, we want Num_{queues} to be greater or equal to Min_{queues}. Therefore, if this is not the case, PMOD decreases the Merging Factor and hence expands the number of queues. From now on, newly arriving chunks will be enqueued in the expanded list of queues.

When the list of queues is consolidated or expanded, care is taken not to create major priority inversion. We give details of the algorithm below.

The SSSP application in the Galois package [26] has a parameter called *delta* that right-shifts the priority values passed to the CPS, hence compressing the range of priorities. This compression can have an effect similar to PMOD's priority merging, but it is an *application-level* change that requires *manual*, per-input tuning, and is static. PMOD attains this effect and the opposite one (i.e., priority expansion) automatically, transparently to application and user, and dynamically.

4.2 PMOD's Priority Ordering

PMOD does not merge or unmerge task queues physically. Instead, it dynamically adjusts the mapping from application-supplied priority values to queues, creating new queues if necessary. To this end, PMOD maintains the base-2 logarithm of the current value of the Merging Factor in variable *lmf*— which stands for Logarithm of Merging Factor. A higher *lmf* means that more priorities are merged. PMOD groups tasks into queues not simply based on the priority value p of the task, but based on the pair (p>>lmf_0, lmf_0) of the task, where >> denotes bitwise right-shift, and lmf_0 is the value of lmf at the time when the task was enqueued. Therefore, to find a queue, PMOD indexes the structure with the pair of values above. To enqueue a task, `Enq` (t, p) inserts task t into the queue (p>>lmf, lmf), creating it if it does not exist.

The algorithm used by PMOD to order queues minimizes priority inversions. Given two queues, (p_1, l_1) and (p_2, l_2), which one has a higher priority? First, PMOD computes $l = \max(l_1, l_2)$. Then, it computes p_1>>$(l - l_1)$ and p_2>>$(l - l_2)$. The lowest value of these two is the queue with the highest priority. Note that by shifting by $l = \max(l_1, l_2)$, PMOD is helping the queue with the tasks with lower l_i, namely those inserted when *lmf* was lower. However, suppose that p_1>>$(l - l_1) = p_2$>>$(l - l_2)$. In this case, the queue with the lower l_i is given the higher priority. Again, tasks inserted when lmf was low are favored.

This ordering algorithm generally prevents low-priority tasks inserted after priority merging from taking precedence over high-priority tasks that were inserted before merging. Consider an example. Assume that *lmf=0* and that we have queues for tasks with priorities {(1,0),(5,0),(8,0),(10,0),(11,0),(32,0)}. Suppose that *lmf* increases to 3, and now a task with $p = 9$ is received to be enqueued. It should be enqueued in queue (9>>3, 3) = (1, 3). Since this queue does not exist, PMOD creates queue (1, 3).

Now, we consider how PMOD orders the new queue with respect to the existing queues. According to the algorithm described above, it orders (1, 3) after (5, 0), since $l = 3$ and (1>>0)>(5>>3). It also orders (1, 3) after queues {(8,0), (10,0), (11,0)} because, while 1>>0 is equal to [8, 10, 11]>>3, its lmf is higher. Finally, it orders (1, 3) before the (32, 0) queue.

Note that with $lmf = 3$, tasks with priorities 8–11 map to the same queue. Thus, the placement of the new queue relative to the existing queues indexed by (8, 0), (10, 0), and (11, 0) is not crucial. What is crucial is to guarantee that the higher priorities (0, 0) to (7, 0) get processed first, by ordering the new queue after their queues. With this ordering, there may be some small priority inversion, but it is tolerable in practice—especially since the priority of the tasks created decreases monotonically during the execution. However, if PMOD only considered p>>lmf, (1, 3) would be placed before (5, 0). The same would be true for any arriving task with $p < 40$ while *lmf=3*. This would create a high priority inversion.

4.3 PMOD Flow

Figure 3 shows the Deq () and Enq () routines. When a Deq () invocation obtains work from the global map rather than obtaining the work locally, we call it a *synchronizing* dequeue. In this case, the *sync_deq* routine is executed. PMOD calls *mergeCheck()* and *unmergeCheck()* to make decisions on merging or unmerging. To make the decisions, PMOD uses some thread-local counters that count a set of events since the last merging/unmerging decision. Such events are the number of Deqs (*nDeqs*), synchronizing Deqs (*nSyncDeqs*), Enqs (*nEnqs*), and the range of priorities enqueued in the system (*minB* and *maxB*). A read of one such counter returns the aggregation of all the thread-local counters. After a merge/unmerge decision, the counters for all the threads are reset. These details are omitted from Figures 3 and 4 for brevity.

4.4 Merging and Unmerging

Figure 4 shows the *mergeCheck()* and *unmergeCheck()* routines. Consider *mergeCheck()* first. Merging is needed when there are too many priority queues and few tasks per queue. PMOD detects this condition by checking if the fraction of Deq calls that go to the global map (i.e., fail to find work locally), *nSyncDeqs/nDeqs*, is greater than $Freq_{global}$. If merging is needed, PMOD computes Num_{queues}, the number of queues needed to cover the priority range observed since the last *lmf* update (Line 4), and *fillRatio*, the average number of tasks that each of these Num_{queues} queues would have received since the last *lmf* update (Line 6). If *fillRatio* is lower than *MinDensity*, PMOD may cautiously increase the Merging Factor, so that Num_{queues} decreases.

```
1  Deq ():          // dequeue routine
2      nDeqs++      // number of dequeue operations since last merge/unmerge
3      if (can dequeue locally )
4          return fast_deq ()
5      else
6          return sync_deq()          // synchronizing dequeue

8  sync_deq ():
9      nSyncDeqs++      // number synchronizing dequeue operations
10     mergeCheck()     // check for, and potentially perform, merging
11     if (lmf not changed)      // if merge didn't occur
12         unmergeCheck()        // check for, and potentially perform, unmerging
13     if (lmf changed)          // if merge or unmerge happened, reset counters
14         nEnqs = nSyncDeqs = nDeqs = 0
15         MaxB = Priority .MIN      // minimum priority value
16         MinB = Priority .MAX      // maximum priority value
17     // rest of the dequeue

19 Enq (task , taskPrio ):          // enqueue routine
20     nEnqs++      // number of enqueue operations since last merge/unmerge
21     // keep track of the priorities created since last merge/unmerge operation
22     MaxB = max(MaxB, taskPrio)
23     MinB = min(MinB, taskPrio)
24     prio = taskPrio >> lmf
25     // proceed to enqueue in queue indexed by prio
```

Figure 3: Deq and Enq routines.

```
1  mergeCheck ():
2    if ((nSyncDeqs / nDeqs) ≤ Freq_global) return
3    // calculate the number of priority groups
4    Num_queues = (MaxB>>lmf) − (MinB>>lmf)
5    // calculate the average number of tasks per priority group
6    fillRatio = nEnqs / Num_queues
7    if ( fillRatio < MinDensity)
8      // may merge priority groups to get closer to MinDensity
9      lmf += log2(MinDensity/ fillRatio )

11 unmergeCheck ():
12   if (nDeqs from single prio_group ≤ MaxPops) return
13   // calculate the number of priority groups
14   Num_queues = (MaxB>>lmf) − (MinB>>lmf)
15   if (Num_queues < Min_queues)
16     // too few prio_groups, may unmerge
17     lmf −= log2(Min_queues / Num_queues)
```

Figure 4: Merge and unmerge operations.

Next, consider *unmergeCheck()* (Figure 4). It is triggered when the number of Deqs from a single priority group since the last *lmf* update exceeds a threshold *MaxPops*. When triggered, the algorithm checks whether Num_{queues} is smaller than threshold Min_{queues}. If so, PMOD may cautiously decrease the Merging Factor.

5 EXPERIMENTAL SYSTEM

5.1 Graph Framework and CPSs

We run our experiments on a 40-core shared-memory machine. The machine has 40 Xeon E7-4860 cores running at 2.27 GHz, organized in 4 sockets of 10 cores each. Each core has 32 KB L1 instruction and data caches, and a 256 KB L2 cache. Each socket has a shared 24 MB L3 cache. The machine has 128 GB of memory.

We evaluate the CPSs using the Galois graph analytics framework [26]. Galois provides a programming model that supports the unordered execution of loop iterations. It executes the iterations in

parallel, treating each iteration as a task. For instance, each iteration can operate on one vertex.

We implement (or use an existing implementation of) the four CPS algorithms described in Section 2.2, plus our proposed `PMOD` CPS described in Section 4. The CPSs are: SprayList (`SL`), MultiQueue (`MQ`); Random-Enqueue Local-Dequeue (`RELD`), `obim` (some applications require variations called `obim-O` and `obim-D` that we describe in Section 5.3), and `PMOD`. Table 3 lists them.

Table 3: CPS algorithms evaluated.

Name	Description
SprayList (`SL`)	Concurrent priority-ordered skip list.
MultiQueue (`MQ`)	Array of concurrent priority queues.
Remote Enqueue, Local Dequeue (`RELD`)	
`obim`	Distributed structure (bag) per priority. For applications that are manually tuned for `obim`, we evaluate both the default and the optimized settings of `obim`, which we call `obim-D` and `obim-O`, respectively.
`PMOD`	Bag per adaptive priority group.

For the SprayList, we use the publicly available implementation by its authors (https://github.com/jkopinsky/SprayList). We base our MultiQueue implementation on the original authors' implementation (obtained by request). In executions with t threads, we use a MultiQueue with $4t$ priority queues. Our implementation replaces the coarse-grained locked sequential priority queues in the original implementation with lock-free skip lists, as we found that the skip lists perform better. We use the skip list implementation from the SprayList code. We implement `RELD` based on our MultiQueue code, to obtain the most accurate comparison. For `obim`, we use the code provided by Galois. We set the chunk size to 64, after tuning experiments.

`PMOD` Parameters. For `PMOD`, we set $Freq_{global}$ to *1/chunk_size*, *MaxPops* to $4 \times$ *chunk_size*, *MinDensity* to 64, and Min_{queues} to 16. For all applications, we start with *lmf*=0.

We select $Freq_{global}$ to be *1/chunk_size* since, if we go to the global map at this frequency, we will have at least one chunk per priority. *MinDensity* is selected to support at least one chunk worth of tasks per priority bin. *MaxPops* and Min_{queues} are selected empirically. *MaxPops* tries to eliminate the case where there are too many tasks per priority, and Min_{queues} sets the minimum number of different priority groups in the system.

5.2 Input Datasets

We evaluate the applications on the input graphs detailed in Table 4. Due to space constraints, however, we only show plots for representative inputs. The input graphs have different characteristics. The USA roads (*rUSA*) and West USA roads (*rW*) graphs are road networks. Twitter40 (*tw*) is a real-world social network graph from Twitter; we use the largest connected component and assign edge weights using a random uniform distribution from the range [0, 100]. Web-google (*wg*) is the web graph released as part of the Google Programming Contest. Soc-LiveJournal1 (*lj*) is the friendship social network of the LiveJournal online community. The *wg* and *lj* datasets come from [19].

Table 4: Input graphs.

Graph	# Vertices	# Edges	Size
USA roads (*rUSA*) [11]	24 M	58 M	628 MB
West USA roads (*rW*) [11]	6 M	15 M	165 MB
Twitter40 (*tw*) [17]	42 M	1469 M	6 GB
Web-Google (*wg*) [22]	875 K	5 M	46 MB
Soc-LiveJournal1 (*lj*) [4]	5 M	69 M	564 MB

5.3 Applications

We evaluate the following applications: Single-Source Shortest Paths (`SSSP`), Breadth-First Search (`BFS`), PageRank (`PR` and `PR-D`), Minimum Spanning Tree (`MST`), and `A*`. All applications but `A*` are standard benchmarks in the Galois distribution; we implement `A*` from scratch.

Single-Source Shortest Paths: The SSSP algorithm in Galois is based on the delta-stepping algorithm [24]. Each task is associated with some vertex v and attempts to extend the shortest path from s to v. The priority of a task is the distance it assigns to its vertex.

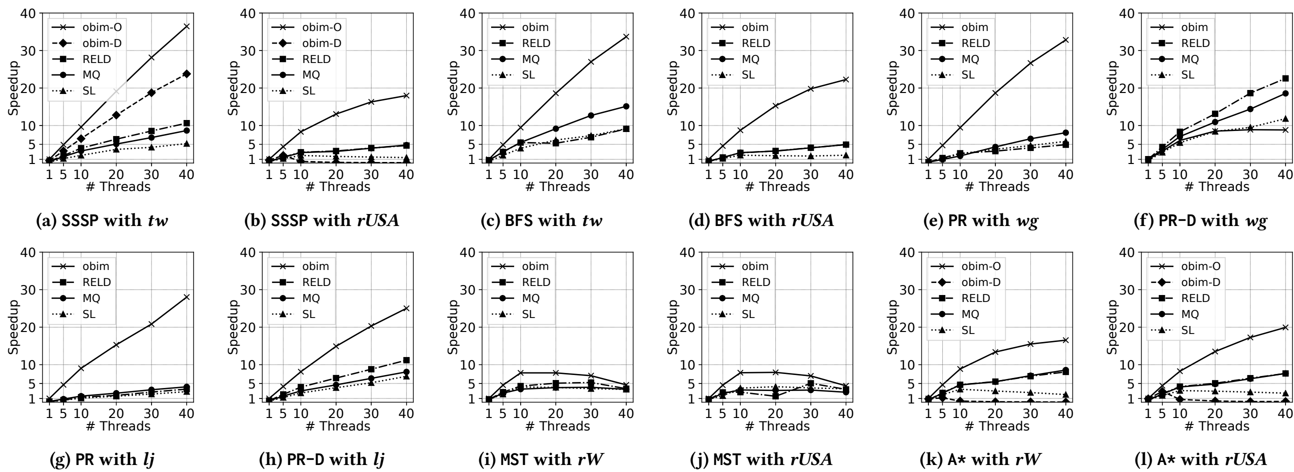
Breadth-First Search: BFS uses breadth-first search to traverse a graph, where the weight of each edge is 1. Tasks are defined as in SSSP, with the priority now being the number of edges on the discovered path.

PageRank: We use a pull-push version of PR, in which the page rank of a vertex is calculated by iterating over its incoming edges (pull) and then propagating the change observed to the vertex's outgoing neighbors (push) [34]. The priority of a task is the PR value of its vertex, which is a floating-point number. To be able to use `obim`, we need to convert the priorities to integers. We evaluate PR with two conversion methods: Taking the whole part of the floating-point number (PR), and taking the whole part plus the three digits after the decimal point (PR-D for "detailed").

Minimum Spanning Tree: MST uses Boruvka's algorithm to find a spanning tree over all vertices with minimum total edge weight. Each task is associated with a vertex. The task picks the vertex's minimum weight edge and merges the vertex and its neighbor connected by the edge, scheduling a task to visit the new vertex. The priority of a task is its vertex's degree.

A*: A* is a path finding algorithm. It calculates the distance from a source vertex s to a destination vertex d. Unlike SSSP, the search is guided by a heuristic value. The heuristic value is the expected distance to vertex d from the currently visited vertex. To guide the search, the priority of a vertex is the sum of its distance to s plus its expected distance to d.

Manually tuned applications: Galois' SSSP application supports manual tuning to obtain the best performance for each input graph. It takes a *delta* (Δ) parameter and right-shifts priority values by Δ bits. This shift decreases the number of distinct priority values and significantly impacts the performance with `obim`. The other CPSs ignore Δ. The value of Δ specified on the Galois web site is 8. However, we search for the values of Δ that attain maximum performance with `obim`. Such optimal values are $\Delta = 14$ for road network graphs, and $\Delta = 0$ for the other graphs. Hence, we evaluate two versions of SSSP: one with the default $\Delta = 8$ (`obim-D`) and one with the optimal Δ value that we identify through empirical search (`obim-O`).



(a) SSSP with *tw* (b) SSSP with *rUSA* (c) BFS with *tw* (d) BFS with *rUSA* (e) PR with *wg* (f) PR-D with *wg*

(g) PR with *lj* (h) PR-D with *lj* (i) MST with *rW* (j) MST with *rUSA* (k) A* with *rW* (l) A* with *rUSA*

Figure 5: Speedups of the CPS schemes relative to the single threaded execution of obim (or obim-O).

Our A* code similarly supports a Δ parameter for scaling priority values with obim, and we evaluate two versions of A*: one with the default $\Delta = 8$ (obim-D) and one with an optimal Δ value (obim-O), which is 14 for all the graphs considered.

6 FINDINGS

6.1 CPS Performance Characteristics

Figure 5 shows the speedups obtained by the CPS schemes as we change the number of threads, relative to the single threaded execution of obim. Recall that for SSSP and A*, we use versions of the applications that are manually tuned for obim, one with the default settings of obim (obim-D), and one with the optimized settings of obim (obim-O). For these two applications, the speedups are relative to obim-O.

The figure shows that, generally, obim or obim-O yield the best performance. However, obim-D often has very poor performance: in SSSP with *rUSA*, A* with *rW*, and A* with *rUSA*, obim-D is the lowest curve, barely above 1. We also see that the speedups vary greatly with the application and input. obim or obim-O attain speedups of 20-40 for many applications and inputs; SL provides the worst average performance; and the distributed CPSs MQ and RELD are in between. We now consider the results in detail.

Search applications (SSSP, BFS, and A*): SSSP's performance under obim heavily depends on the input and on the Δ parameter. For instance, SSSP obtains nearly linear speedup under obim-O on the *tw* input (Figure 5a). Under obim-D, the speedup is lower, about 24 at 40 threads, but still higher than under the other CPSs. On the other hand, on the *rUSA* input, the speedup under obim-O does not exceed 20, and under obim-D, the speedup collapses and the application runs slower in parallel than sequentially (Figure 5b). As we show later, this collapse is due to the lack of sufficient work per priority value, as described in observation *O2* (Section 3.3). In contrast, while BFS shows some input-sensitivity under obim, it is less drastic. Under *rUSA*, BFS still sees a significant speedup of 22. While this is less than the speedup obtained for *tw*, the performance of BFS on *rUSA* does not collapse (Figures 5c–5d). Finally, A*, which runs only on the road networks, behaves similarly to SSSP. Specifically, obim-O yields the best speedup, but the performance collapses with obim-D (Figures 5k–5l).

PR and PR-D: Recall that the difference between PR and PR-D is that the latter has a 1,000× wider range of priorities. Such change has a major effect on obim. Specifically, PR with *wg* under obim yields a speedup of 32 for 40 cores, making obim the best CPS (Figure 5e). However, PR-D under obim becomes substantially slower. On PR-D, all the other CPSs do better than obim, which delivers a speedup lower than 10 (Figure 5f). This effect is also due to observation *O2* in Section 3.3. Although not shown because the figure shows speedups relative to obim, the other CPSs are much less affected by the range of priorities. In summary, obim needs many tasks per priority value to perform well, whereas the remaining CPSs are much less sensitive to this metric.

MST: Unlike the other applications, MST does not scale past a single socket (10 threads) under any CPS. Under obim, MST enjoys almost linear scalability within the socket, but subsequently degrades and obtains a speedup of 5 at 40 threads (Figures 5i–5j). Under the other CPSs, the speedup never exceeds 5 even within a socket. The reason for MST's lack of scalability is that, unlike the other applications, MST merges vertices as it executes, thereby decreasing the number of vertices and the available parallelism [13].

6.2 CPS Performance Analysis & Observations

We now analyze the reasons for the CPS performance trends and empirically support the high-level observations made in Section 3. Figure 6 breaks down the 40-threaded execution time of the applications under the different CPS schemes. We use the categories detailed in Table 1: performing *useless* work (*UWork*), performing *good* work (*GWork*), executing CPS Enq (*Enq*), executing CPS Deq that returns a task (*Deq*), executing CPS Deq that fails to return a task (*FDeq*), and executing non-CPS Galois framework code (*Other*).

6.2.1 CPS Overhead Determines the Execution Time. From Figure 6, we see that the CPS overhead (*Enq*, *Deq*, and *FDeq*), rather than useless work, typically determines the execution time. Generally, the relaxed priority scheduling performed by the CPSs creates negligible useless work. Only obim sometimes creates non-negligible

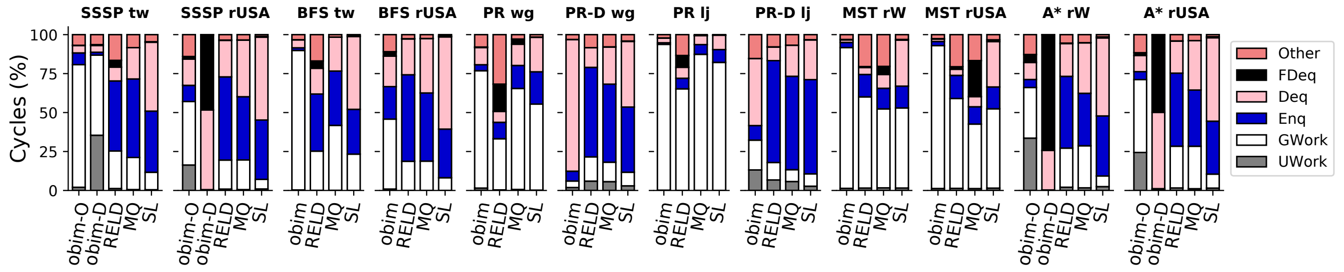


Figure 6: Breakdown of the normalized execution cycles for 40-threaded executions.

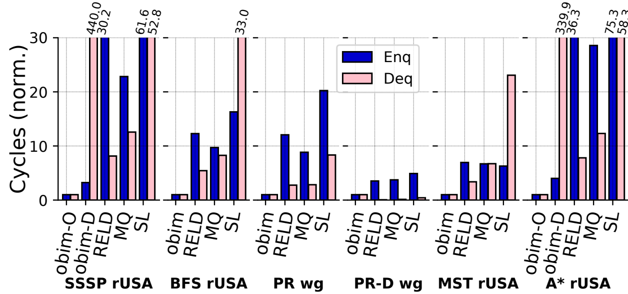


Figure 7: Cycles per Enq and Deq operation for 40-threaded executions normalized to obim (or obim-O).

useless work, which we explore shortly. For the most part, applications under obim spend less than 10% of their time inside the CPS (*Enq*, *Deq*, or *FDeq*). The exceptions are PR-D and the search applications with the road networks, which we discuss shortly. With the other CPSs, applications typically spend 50%–90% of their time in CPS code.

The differences in CPS time are explained by Figure 7, which shows the cycles per Enq and Deq operations. Observe that obim's Enq and Deq operations are orders of magnitude cheaper, on average, than in the other CPSs. The main factor behind this difference is that obim buffers both enqueued and dequeued tasks in *chunks*, thereby amortizing communication costs by improving cache locality.

Another factor behind CPS time is data structure complexity. First, chunk insertion and removal from obim's FIFO queues are $O(1)$ operations, whereas searching a skip list is an $O(\log n)$ operation, where n is the size of the skip list. Second, MQ and RELD distribute tasks among multiple skip lists, resulting in shorter skip lists than the single skip list maintained in SL. Thus, Enqs in SL are slower (Figure 7). Finally, removing a task in MQ and RELD is an $O(1)$ operation (removing the head of some skip list), whereas SL performs a random walk and is thus slower (Figure 7).

As obim is the most competitive CPS, we now study its performance in detail.

6.2.2 Performance Sensitivity to Priority Values (O2). Performance with obim decreases when there are few tasks per priority. In this case, threads that enqueue tasks do not manage to fill their chunks. Since chunks are thread-local, such tasks remain invisible to other threads. Threads spend substantial time going over many priority bags trying to find work. This case manifests itself as higher Deq time in the application, and more cycles per Deq operation. Moreover, in some cases, threads do not find work at all, even though there are private tasks pending execution, leading to load imbalance. *FDeq* cycles capture such unsuccessful dequeue attempts.

The search applications on the road networks and PR-D experience this effect. For example, the road networks' edge weights are drawn from large ranges. There are many priority bags, with on average 1.5 tasks per priority. Figure 6 shows the result of this effect. Under obim-D, SSSP on *rUSA*, and A* on *rW* and *rUSA* spend all of their time searching for tasks to execute. The combination of *Deq* and *FDeq* cycles accounts for all the execution cycles. Note that *FDeq*—which is small in all other CPSs and inputs—can be over 50% of the execution time. A similar effect is shown in Figure 6 for obim in PR-D with *wg* and *lj*.

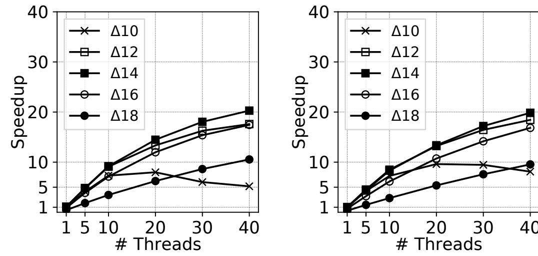
6.2.3 Useless Work vs. CPS Efficiency Tradeoffs (O1). The Δ parameter in SSSP and A* right-shifts priority values by some bits, effectively *compressing* the priorities into fewer bags. Increasing Δ is thus a tradeoff. It increases the average number of tasks per priority bag, which makes chunking more effective and helps obim find tasks faster. However, lumping together tasks with highly different priorities increases the chance of running tasks out of priority order and performing useless work. Figure 6 shows this tradeoff. For SSSP on *rUSA*, and A* on *rW* and *rUSA*, obim-O (which uses Δ =14) reduces the fraction of the execution in the CPS (*Enq*, *Deq*, and *FDeq*) to 25% or less. This is compared to ≈100% in obim-D. However, 20–35% of the time in obim-O is now spent on useless work. Still, obim-O is so lightweight that, despite executing useless work, it is faster than SL, MQ, and RELD (Figures 5b,5k and 5l). These CPSs have little useless work (Figure 6), but their overhead is so high that the work quality becomes a second-order effect.

Useless work is not free, however. When the number of tasks per priority is large, avoiding useless work pays off. For instance, for SSSP on the *tw* input, running with obim-D (Δ = 8) leads to 35% of the time being spent on useless work, and a maximum speedup of about 24, whereas with obim-O (Δ = 0), useless work is negligible and speedup is nearly linear.

To study this tradeoff, we consider SSSP and A* (which use the Δ parameter), and vary Δ. Figure 8 shows the speedups of SSSP and A* for *rUSA* (*rW* shows similar behavior and is omitted due to space constraints), as we vary Δ from 10 to 18. The speedups are normalized to single-threaded runs with Δ = 10. Figure 9, on the other hand, shows detailed cycle breakdowns similar to Figure 6.

From Figure 8, we see that the highest speedups are attained with Δ = 14 (the optimal value used in Section 6.1). As Δ moves higher or lower than 14, the speedups decrease. While Δ = 12 and Δ = 16 deliver acceptable speedups, values further out do not. For example, with Δ = 10, the speedups at 40 threads are 5 and 8.

To understand this behavior, consider the cycle breakdown in Figure 9. The figure corresponds to 40-threaded executions of SSSP



(a) SSSP with *rUSA* (b) A* with *rUSA*

Figure 8: Speedups of obim for different Δ values for SSSP and A*, relative to the single-threaded execution with Δ = 10.

and A*. When Δ is below optimum, there are many priority bags and few tasks per bag. Many tasks remain invisible, buffered in non-filled thread-local chunks. Consequently, idle threads cannot find work efficiently. As shown in the Δ = 10 bars, *Deq* and *FDeq* consume the large majority of cycles. As Δ increases, the contribution of *Deq* and *FDeq* decrease, but useless work appears. When Δ is above the optimum, many different priorities are placed in the same bag, increasing the useless work.

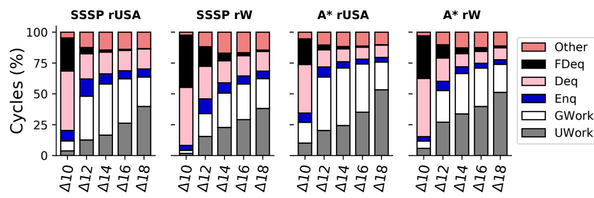
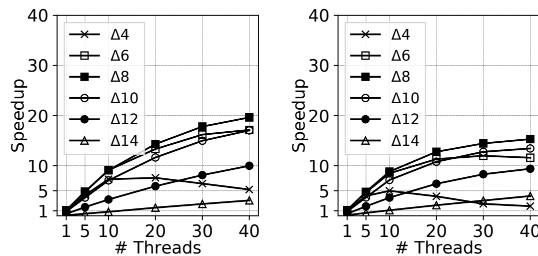


Figure 9: Breakdown of execution cycles for 40 threaded-executions of SSSP and A*, while varying Δ.

6.2.4 *Optimal Priority Merging Depends on Input Data (O2).* How aggressively priorities need to be compressed for obim to attain good performance depends on the priority value distribution *rather than the graph structure*. To show this, we repeat the above experiments scaling down all the edge weights in the graph by 64. This change does not alter shortest paths or the graphs' topology, but changes the range of priority values. Figure 10 shows speedups for SSSP running *rUSA* and *rW* for different Δ values. We can see that now, the optimal Δ decreases to 8.

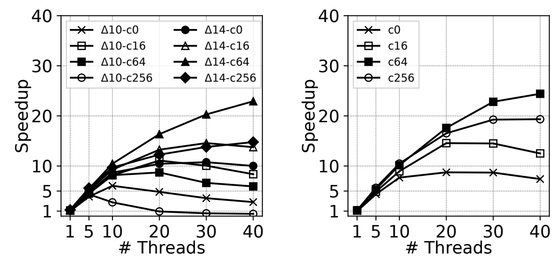


(a) SSSP with *rUSA* (b) SSSP with *rW*

Figure 10: Speedups of obim for different Δ values for SSSP with edge weights scaled down by 64.

6.2.5 *Amortizing Communication Using Chunking (O3).* Processing tasks in chunks reduces obim's Enq and Deq operation cost to $O(1)$ on average. For Enq and Deq, obim accesses shared data structures only once in c operations, where c is the chunk size. Here, we show that chunking is an important factor in obim's performance, and analyze the interaction of chunking and priority compression.

Figure 11 shows the speedups of SSSP and BFS for *rUSA* as we vary the chunk size from 0 (*c0*, chunking disabled) to 256 (*c256*). *rW* shows similar behavior and is omitted due to space constraints. For SSSP, we have curves for Δ equal to 10 and 14. The speedups are normalized to single-threaded runs with no chunking and, for SSSP, Δ = 10. We see that the highest speedups are attained with the default chunk size of 64 for both SSSP (with the optimal Δ of 14) and BFS. Both larger and smaller chunk sizes decrease the speedups. For example, with chunking disabled, the speedups at 40 threads are 2-3 times lower.



(a) SSSP with *rUSA* (b) BFS with *rUSA*

Figure 11: Speedups of obim for different chunk sizes for SSSP and BFS, relative to the single-threaded execution with no chunking (and Δ = 10 for SSSP).

To understand this behavior, consider the cycle breakdown of the 40-threaded executions in Figure 12. Without chunking, obim accesses shared structures on each Enq and Deq operation. Hence, the *Enq* and *Deq* categories account for ≈80-90% of the cycles. As we increase the chunk size, this fraction goes down. However, larger chunks are harder to fill. Tasks thus remain buffered and inaccessible to other threads. This causes other threads to either work on low-quality tasks (*UWork*) or fail to find tasks altogether (*FDeq*).

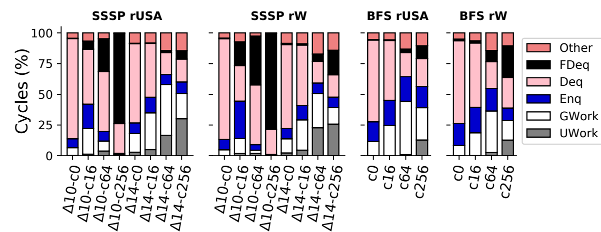
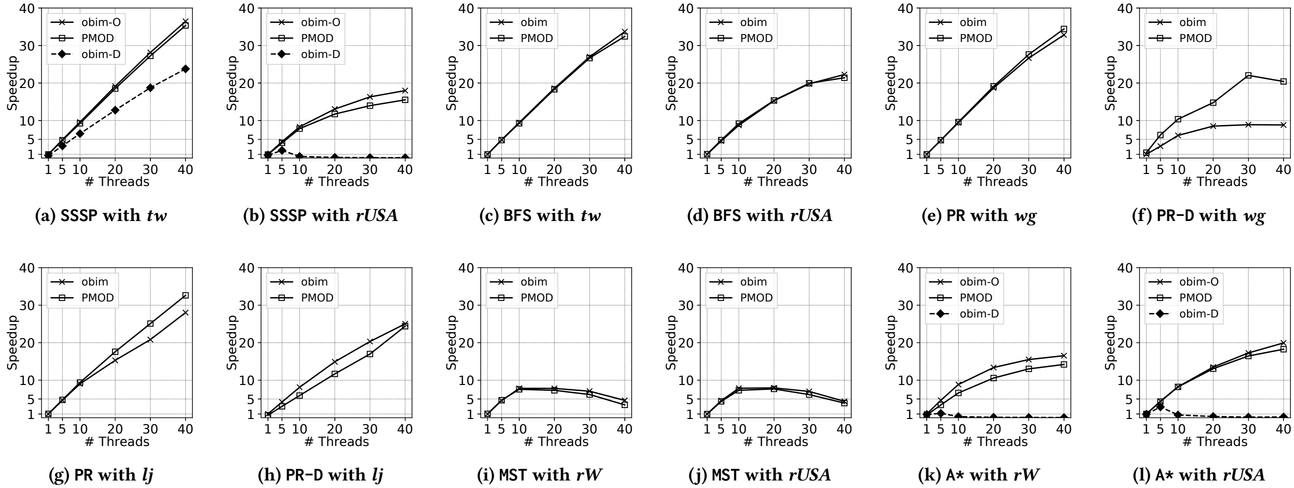


Figure 12: Breakdown of execution cycles for 40 threaded-executions of SSSP and BFS while varying the chunk size. For SSSP, we show bars with the suboptimal Δ = 10 and the optimal Δ = 14.

6.2.6 *Chunk Size: Load Balancing vs Overhead Tradeoff.* A main reason for the inefficiency with the small suboptimal Δ equal to 10 is that, with few tasks per priority bag, chunks do not get filled and thus tasks remain invisible to other threads. Now we evaluate whether decreasing the chunk size solves this problem. Figure 11a shows the speedups of SSSP using Δ = 10 with varying chunk size. A smaller chunk size of 16 yields the best performance, instead of 64. Small 16-entry chunks fill faster and become visible faster,



(a) SSSP with *tw* (b) SSSP with *rUSA* (c) BFS with *tw* (d) BFS with *rUSA* (e) PR with *wg* (f) PR-D with *wg*

(g) PR with *lj* (h) PR-D with *lj* (i) MST with *rW* (j) MST with *rUSA* (k) A* with *rW* (l) A* with *rUSA*

Figure 13: Speedups of PMOD and obim variants relative to the single-threaded execution of obim (or obim-O).

which almost eliminates *UWork* cycles, but they impose high Enq overhead (Figure 12). Consequently, the speedup with 16-entry chunks is only 30% better than with the default of 64, and the best execution time with suboptimal $\Delta = 10$ remains $\approx 3\times$ slower than with the optimal $\Delta = 14$ and chunk size of 64.

6.2.7 Summary of Findings. With the right amount of priority compression, chunks of size 64 serve well to amortize communication without hurting load balancing, whereas adjusting chunk size does not compensate for suboptimal priority compression. The optimal level of priority compression depends on the priority distribution, and cannot be determined statically or based on graph topology. This motivates our proposed PMOD CPS.

6.3 Effectiveness of PMOD

6.3.1 PMOD Speedups. Table 5 presents an overall comparison of CPSs at 40 cores. In the first row, we pick a given CPS and, for each application, compute the speedup of that CPS over the best CPS for that application. We then take the geometric mean over all the applications. The resulting number indicates how close that CPS is to being the optimal choice as the default CPS for all applications. PMOD's value of 0.93—the highest among all CPSs—means that PMOD is always comparable to the application-specific best performer. Only obim-O has a similar number, but it is not a viable choice for a default CPS since it requires extensive, workload-specific, manual tuning to achieve this result.

Table 5: Geometric mean of the CPS speedup compared to best CPS for each application, and geometric mean of speedup for 40 cores.

	PMOD	obim-D	obim-O	RELD	MQ	SL
W.r.t best CPS for the app.	0.93	0.55	0.89	0.35	0.34	0.18
Speedup for 40 cores	17	10.6	17	6.7	6.5	3.6

The second row shows the geometric mean speedup of each CPS over a single-threaded obim execution. PMOD is much better than the CPSs that do not require manual tuning. The same is true for obim-O, but obim-O requires manual tuning.

We further evaluate PMOD by comparing its execution time to the obim variants. Figure 13 shows the speedups of the applications under PMOD, obim, and, when applicable, obim-O and obim-D. All curves are relative to the single-threaded execution time of obim (or obim-O, when applicable).

We see that PMOD performs as well as obim (and obim-O when applicable) in all cases. Recall that obim-O is obtained through manual tuning, searching for and identifying the optimal Δ. What makes PMOD attractive is its ability to match obim-O's performance without any tuning.

In fact, PMOD outperforms obim in PR-D on the *wg* input. PMOD is twice as fast as obim for 40 threads. The reason is that, as discussed in Section 6.2, obim does not work well with the large range of priority values in PR-D. Instead of having the programmer manually work around this problem, as was done in SSSP with the Δ parameter, PMOD adapts to the observed priority ranges and successfully speeds up the application, obtaining a 20× speedup.

To gain further insights, Figure 14 breaks down the 40-threaded execution time of the applications in these experiments. For the most part, the breakdowns in PMOD are very similar to those in obim. Importantly, when we have obim-O and obim-D bars, PMOD is similar to obim-O, while obim-D has either many *Deq* and *FDeq* cycles, or many *UWork* cycles. In the case of PR-D on the *wg* input, PMOD has a higher fraction of *GWork* cycles than obim.

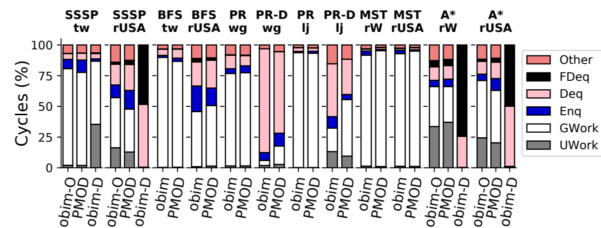


Figure 14: PMOD vs. obim execution cycle breakdown (40 threads).

6.3.2 PMOD Dynamics. We now illustrate how PMOD adjusts its priority merging over time. When an application starts, lmf is

zero. Table 6 shows the sequence of values that lmf takes as our applications execute. We show data for each application and dataset shown in Figure 13. The table shows the final value of lmf, the number of changes, and the values that lmf takes.

Table 6: lmf dynamics. The table shows the final lmf values, the number of lmf changes, and the sequence of actual lmf values.

App.	Dataset	Final lmf	# Changes	lmf Values
SSSP	*tw*	0	0	0
	rUSA	13	2	0-12-13
BFS	*tw*	0	0	0
	rUSA	2	1	0-2
PR	*wg*	3	2	0-1-3
	lj	2	2	0-1-2
PR-D	*wg*	10	1	0-10
	lj	9	2	0-1-9
MST	*rW*	3	2	0-2-3
	rUSA	3	3	0-1-2-3
A*	*rW*	16	1	0-16
	rUSA	13	2	0-12-13

The data shows that lmf increases monotonically. For SSSP with *rUSA* and A*, where under obim-O we manually set Δ to 14, PMOD converges to *lmf* values of 13 and 16, which are close to the optimal Δ. As a result, PMOD's performance is close to obim-O's. Moreover, applications such as PR-D, which do not use Δ, benefit from PMOD's merging mechanism. For instance, PMOD automatically sets the *lmf* value for PR-D to 9 and 10.

lmf often goes through multiple changes. The timing of the changes differs across applications, datasets, and number of threads. Often, the merge operations occur in the beginning of the execution, during the first 1% of Deq operations. For example, this happens in SSSP with the *rUSA* input. In this case, PMOD quickly increases *lmf* first to 12, and then to 13, which becomes its final value. However, this is not always the case. For instance, in MST, the first merge occurs only when around 40% of Deq operations have executed.

The number of threads also affects merging. We compare the time of the first merge operation in executions with 40 threads and with 1 thread. Although not shown in Table 6, we find that for SSSP on *rUSA* with 1 thread, only about 60 Deq operations execute before the first merge operation. For SSSP with 40 threads, the first merge only occurs after 15-20 K Deq operations. For MST, the behavior is different. In both single- and 40-threaded executions, the first merge operation occurs when around 40% of the Deq have executed.

7 IMPLICATIONS ON COMPUTER ARCHITECTURES

Our analysis provides insights into the bottlenecks of concurrent priority scheduling for graph algorithms in large servers. It is sobering to see that sophisticated skip list-based CPSs are overwhelmed by enqueueing and dequeueing overheads. This is despite employing scalable data structures that perform searches in parallel without synchronization, and avoid synchronization hotspots in updates. We do not believe that hardware support in large NUMA servers should focus on improving synchronization for such CPS designs. Instead, it should focus on improving the obim and PMOD approaches to CPS.

PMOD typically devotes a large fraction of cycles to *GWork*, as shown in Figure 14. However, the figure also shows that there are still some cases where *GWork* is a small fraction of the total cycles. PMOD is still sometimes a victim of the fundamental CPS tradeoff: either it suffers from *Deq*/*FDeq* cycles, or from *UWork*. We need to replace PMOD software structures with hardware that frees PMOD from this tradeoff. One example is hardware to make partially-full chunks quickly available to idle threads. Another is fast communication of the work list and the partially-full work list across sockets.

8 RELATED WORK

Several graph analytics frameworks [9, 21, 26, 31, 32, 37] have been developed for shared-memory machines. Ligra [31] abstracts away graph traversals through mapping computations over a subset of vertices or edges in parallel. Julienne [9] builds upon Ligra by grouping together similar graph entities, such as vertices, edges, or other graph motifs, into buckets.

Many concurrent priority queues [2, 7, 29, 35, 36] have been introduced for task-based priority scheduling. Techniques such as Flat Combining [14] and Elimination [25] are adopted by Calciu et al. [7] to reduce enqueue/dequeue overheads without compromising on priority constraints. In contrast, relaxed priority schedulers [2, 26, 29, 35, 36] trade off priority constraints for lower synchronization. Lenharth et al. studied the performance of priority queues as graph analytics CPSs [18]. However, they did not consider relaxed priority queues like SprayList or MultiQueues.

There has been much work on algorithm-specific optimizations of different graph problems, e.g., for SSSP [23], BFS [5], and MST [16]. However, our focus is on optimizing generic graph frameworks and not on targeted optimizations.

9 CONCLUSION

Graph processing frameworks use CPSs to execute tasks largely according to their priority order. CPSs are performance critical, but there has been little insight on the relative strengths and weaknesses of the different CPS designs. We addressed this question with a detailed empirical performance analysis of four state-of-the-art representative CPS designs on a 40-core shared-memory machine. We observed that in all CPSs but obim, the overall cost of enqueueing and dequeueing is typically higher than the task execution time. This is despite employing scalable data structures. Further, the obim CPS, which is designed to reduce enqueueing and dequeueing overheads at the expense of sometimes executing useless work, also has limitations. While it typically performs best, it leads to significant slowdowns under some priority distributions. With these insights, we developed the new PMOD CPS. It is based on the obim approach but dynamically adapts to the ranges of priorities exhibited by the application. PMOD is robust and delivers the best performance overall.

ACKNOWLEDGMENTS

This research was funded in part by the National Science Foundation (grants CNS 1763658 and CCF 1725734), and the Israel Science Foundation (grant 2005/17).

REFERENCES

[1] Dan Alistarh, Justin Kopinsky, Jerry Li, and Giorgi Nadiradze. 2017. The Power of Choice in Priority Scheduling. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC '17)*. ACM, New York, NY, USA, 283–292. https://doi.org/10.1145/3087801.3087810

[2] Dan Alistarh, Justin Kopinsky, Jerry Li, and Nir Shavit. 2015. The SprayList: A Scalable Relaxed Priority Queue. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2015)*. ACM, New York, NY, USA, 11–20. https://doi.org/10.1145/2688500.2688523

[3] Nina Amenta, Sunghee Choi, and Günter Rote. 2003. Incremental Constructions Con BRIO. In *Proceedings of the Nineteenth Annual Symposium on Computational Geometry (SCG '03)*. ACM, New York, NY, USA, 211–219. https://doi.org/10.1145/777792.777824

[4] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. 2006. Group Formation in Large Social Networks: Membership, Growth, and Evolution. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '06)*. ACM, New York, NY, USA, 44–54. https://doi.org/10.1145/1150402.1150412

[5] Scott Beamer, Krste Asanović, and David Patterson. 2012. Direction-optimizing Breadth-first Search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12)*. IEEE Computer Society Press, Los Alamitos, CA, USA, Article 12, 10 pages. http://dl.acm.org/citation.cfm?id=2388996.2389013

[6] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. 2012. Internally Deterministic Parallel Algorithms Can Be Fast. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '12)*. ACM, New York, NY, USA, 181–192. https://doi.org/10.1145/2145816.2145840

[7] Irina Calciu, Hammurabi Mendes, and Maurice Herlihy. 2014. The Adaptive Priority Queue with Elimination and Combining. In *Distributed Computing*, Fabian Kuhn (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 406–420.

[8] Boris V. Cherkassy and Andrew V. Goldberg. 1995. On Implementing Push-Relabel Method for the Maximum Flow Problem. In *Proceedings of the 4th International IPCO Conference on Integer Programming and Combinatorial Optimization*. Springer-Verlag, London, UK, UK, 157–171. http://dl.acm.org/citation.cfm?id=645586.659457

[9] Laxman Dhulipala, Guy Blelloch, and Julian Shun. 2017. Julienne: A Framework for Parallel Graph Algorithms Using Work-efficient Bucketing. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '17)*. ACM, New York, NY, USA, 293–304. https://doi.org/10.1145/3087556.3087580

[10] E. W. Dijkstra. 1959. A Note on Two Problems in Connexion with Graphs. *Numer. Math.* 1, 1 (Dec. 1959), 269–271. https://doi.org/10.1007/BF01386390

[11] DIMACS. 2006. 9th DIMACS Implementation Challenge. Retrieved August 28, 2019 from http://www.dis.uniroma1.it/challenge9/download.shtml

[12] Keir Fraser. 2004. *Practical lock-freedom.* Ph.D. Dissertation. University of Cambridge, Computer Laboratory.

[13] Muhammad Amber Hassaan, Martin Burtscher, and Keshav Pingali. 2011. Ordered vs. Unordered: A Comparison of Parallelism and Work-efficiency in Irregular Algorithms. *SIGPLAN Not.* 46, 8 (Feb. 2011), 3–12. https://doi.org/10.1145/2038037.1941557

[14] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. 2010. Flat Combining and the Synchronization-parallelism Tradeoff. In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '10)*. ACM, New York, NY, USA, 355–364. https://doi.org/10.1145/1810479.1810540

[15] M. C. Jeffrey, S. Subramanian, C. Yan, J. Emer, and D. Sanchez. 2015. A scalable architecture for ordered parallelism. In *48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 228–241. https://doi.org/10.1145/2830772.2830777

[16] Seunghwa Kang and David A. Bader. 2009. An Efficient Transactional Memory Algorithm for Computing Minimum Spanning Forest of Sparse Graphs. *SIGPLAN Not.* 44, 4 (Feb. 2009), 15–24. https://doi.org/10.1145/1594835.1504182

[17] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a Social Network or a News Media?. In *Proceedings of the 19th International Conference on World Wide Web (WWW '10)*. ACM, New York, NY, USA, 591–600. https://doi.org/10.1145/1772690.1772751

[18] Andrew Lenharth, Donald Nguyen, and Keshav Pingali. 2015. Priority Queues Are Not Good Concurrent Priority Schedulers. In *Euro-Par 2015: Parallel Processing*, Jesper Larsson Träff, Sascha Hunold, and Francesco Versaci (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 209–221.

[19] Jure Leskovec and Rok Sosič. 2016. SNAP: A General-Purpose Network Analysis and Graph-Mining Library. *ACM Transactions on Intelligent Systems and Technology (TIST)* 8, 1 (2016), 1.

[20] Jonatan Lindén and Bengt Jonsson. 2013. A Skiplist-Based Concurrent Priority Queue with Minimal Memory Contention. In *Proceedings of the 17th International Conference on Principles of Distributed Systems - Volume 8304 (OPODIS 2013)*. Springer-Verlag, Berlin, Heidelberg, 206–220. https://doi.org/10.1007/978-3-319-03850-6_15

[21] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph Hellerstein. 2010. GraphLab: A New Framework for Parallel Machine Learning. In *Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence (UAI'10)*. AUAI Press, Arlington, Virginia, United States, 340–349. http://dl.acm.org/citation.cfm?id=3023549.3023589

[22] Michael W. Mahoney, Anirban Dasgupta, Kevin J. Lang, and Jure Leskovec. 2009. Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters. *Internet Mathematics* 6, 1 (2009), 29–123.

[23] Saeed Maleki, Donald Nguyen, Andrew Lenharth, María Garzarán, David Padua, and Keshav Pingali. 2016. DSMR: A Parallel Algorithm for Single-Source Shortest Path Problem. In *Proceedings of the 2016 International Conference on Supercomputing (ICS '16)*. ACM, New York, NY, USA, Article 32, 14 pages. https://doi.org/10.1145/2925426.2926287

[24] U. Meyer and P. Sanders. 2003. Δ-stepping: A Parallelizable Shortest Path Algorithm. *J. Algorithms* 49, 1 (Oct. 2003), 114–152. https://doi.org/10.1016/S0196-6774(03)00076-2

[25] Mark Moir, Daniel Nussbaum, Ori Shalev, and Nir Shavit. 2005. Using Elimination to Implement Scalable and Lock-free FIFO Queues. In *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '05)*. ACM, New York, NY, USA, 253–262. https://doi.org/10.1145/1073970.1074013

[26] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A Lightweight Infrastructure for Graph Analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 456–471. https://doi.org/10.1145/2517349.2522739

[27] Donald Nguyen and Keshav Pingali. 2011. Synthesizing Concurrent Schedulers for Irregular Algorithms. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, New York, NY, USA, 333–344. https://doi.org/10.1145/1950365.1950404

[28] William Pugh. 1990. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Commun. ACM* 33, 6 (June 1990), 668–676. https://doi.org/10.1145/78973.78977

[29] Hamza Rihani, Peter Sanders, and Roman Dementiev. 2015. Brief Announcement: MultiQueues: Simple Relaxed Concurrent Priority Queues. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '15)*. ACM, New York, NY, USA, 80–82. https://doi.org/10.1145/2755573.2755616

[30] N. Shavit and I. Lotan. 2000. Skiplist-based concurrent priority queues. In *Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000*. 263–268. https://doi.org/10.1109/IPDPS.2000.845994

[31] Julian Shun and Guy E. Blelloch. 2013. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '13)*. ACM, New York, NY, USA, 135–146. https://doi.org/10.1145/2442516.2442530

[32] Jiawen Sun, Hans Vandierendonck, and Dimitrios S. Nikolopoulos. 2017. GraphGrind: Addressing Load Imbalance of Graph Partitioning. In *Proceedings of the International Conference on Supercomputing (ICS '17)*. ACM, New York, NY, USA, 16:1–16:10. https://doi.org/10.1145/3079079.3079097

[33] Håkan Sundell and Philippas Tsigas. 2005. Fast and Lock-free Concurrent Priority Queues for Multi-thread Systems. *J. Parallel Distrib. Comput.* 65, 5 (May 2005), 609–627. https://doi.org/10.1016/j.jpdc.2004.12.005

[34] Joyce Jiyoung Whang, Andrew Lenharth, Inderjit S. Dhillon, and Keshav Pingali. 2015. Scalable Data-Driven PageRank: Algorithms, System Issues, and Lessons Learned. In *Euro-Par 2015: Parallel Processing*, Jesper Larsson Träff, Sascha Hunold, and Francesco Versaci (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 438–450.

[35] Martin Wimmer, Jakob Gruber, Jesper Larsson Träff, and Philippas Tsigas. 2015. The Lock-free k-LSM Relaxed Priority Queue. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2015)*. ACM, New York, NY, USA, 277–278. https://doi.org/10.1145/2688500.2688547

[36] Martin Wimmer, Francesco Versaci, Jesper Larsson Träff, Daniel Cederman, and Philippas Tsigas. 2014. Data Structures for Task-based Priority Scheduling. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '14)*. ACM, New York, NY, USA, 379–380. https://doi.org/10.1145/2555243.2555278

[37] Kaiyuan Zhang, Rong Chen, and Haibo Chen. 2015. NUMA-aware Graph-structured Analytics. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2015)*. ACM, New York, NY, USA, 183–193. https://doi.org/10.1145/2688500.2688507