# Multi Bucket Queues: Efficient Concurrent Priority Scheduling

Guozheng Zhang
University of Toronto
Canada
guozheng.zhang@mail.utoronto.ca

Gilead Posluns
University of Toronto
Canada
gil.posluns@mail.utoronto.ca

Mark C. Jeffrey
University of Toronto
Canada
mcj@ece.utoronto.ca

## ABSTRACT

Many irregular algorithms converge more quickly when they execute tasks in a specific order. When this order is discovered at run time, the algorithm demands a dynamic task scheduler. Scaling a priority scheduler to large systems with many cores is challenging and while many concurrent priority schedulers (CPS) have been proposed, a general classification of their design space is still lacking. We survey prior work and propose three dimensions for the design of CPSs: the *degree of synchrony*, the *drift of priorities*, and the *underlying data structure*. We use this taxonomy to classify existing schedulers and evaluate their strengths and weaknesses.

Building on our observations, we propose the *Multi Bucket Queue* (MBQ) which targets a promising unexplored point in the design space for concurrent priority scheduling. The MBQ leverages the strengths of the MultiQueue and Multi-Level Bucket Queue, while avoiding their weaknesses, yielding a CPS that keeps threads busy and running useful work, yet with high-efficiency queue operations. Our experimental results show that the MBQ is competitive with or outperforms prior work.

## CCS CONCEPTS

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

## KEYWORDS

concurrent priority scheduling; relaxed algorithms; bucket queue

**ACM Reference Format:**
Guozheng Zhang, Gilead Posluns, and Mark C. Jeffrey. 2024. Multi Bucket Queues: Efficient Concurrent Priority Scheduling. In *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '24), June 17–21, 2024, Nantes, France.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3626183.3659962

## 1 INTRODUCTION

Priority scheduling is critical to efficiently solve important problems in domains such as graph processing, statistical inference, numerical optimization, among others. *Ordered* algorithms for such problems comprise *tasks* that execute in some (partial) priority order to converge faster and reduce work compared to *unordered* algorithms [38]. For example, the unordered Bellman-Ford algorithm solves the single-sourced shortest paths (`sssp`) problem in $O(|V||E|)$ time, while the ordered Dijkstra's algorithm with a heap takes $O(|V| \log |V|+|E|)$. Other algorithms that benefit from priority scheduling include greedy approximate set cover (`sc`) [28], deterministic maximal independent set (`mis`) [5], push-based PageRank (`pr`) [53] and residual belief propagation (`rbp`) [19].

The efficiency of priority scheduling is at odds with scalability on manycore architectures with tens to hundreds of cores. Scheduling tasks to threads with a global priority queue sacrifices parallelism to contention [31]. Instead, prior concurrent priority schedulers broadly fall under two categories with disparate benefits and drawbacks: *synchronous* [17, 60] or *asynchronous* [7, 31, 36, 40, 42, 58].

Synchronous priority scheduling breaks execution into *supersteps* where equal priority tasks run concurrently between barriers. With monotonic priorities, synchronous scheduling follows a *strict* priority order that is equivalent to sequential execution, enabling *work-efficient* scheduling. With unordered tasks per superstep, synchronous schedulers employ efficient data structures with good spatial locality like bucket queues [15, 17, 60]. Unfortunately, barriers limit parallelism as threads idle, waiting for stragglers to finish the superstep. The more available cores, the more aggregate cycles wasted due to just one straggler. At the extreme, execution is sequential in supersteps with only a few tasks between barriers.

Asynchronous priority scheduling allows concurrent execution of tasks *across priority values* to eliminate idling. Asynchronous scheduling may retain the strict priority order with *speculative* task-level parallelism [27, 29]. The system speculates that tasks can safely execute out of priority order but rolls back their effects upon a detected misspeculation. Unfortunately, the benefits of work efficiency (when aborts are rare) are overwhelmed by the overheads of speculation in software [12, 25, 26]. Asynchronous scheduling without speculation provides only a *relaxed* priority order: a best effort to dispatch tasks to threads in priority order.

Asynchronous relaxed priority scheduling keeps threads busy and avoids speculation overheads but introduces a tradeoff between work inefficiency and data structure contention/computational complexity. To limit the *priority drift* [46] and therefore reduce the amount of redundant work, some prior schedulers maintain what we call a *global ordering*. These schedulers provide probabilistic guarantees on the relative rank of a popped task compared to all other tasks in the scheduler [7, 40, 42]. This keeps the parallel priority schedule close to the sequential priority schedule. Unfortunately, maintaining this global ordering demands frequent communication between threads and the scheduler and prior work has employed expensive, irregular data structures, such as heaps and skip lists, to maintain the rank guarantees [7, 20, 21, 40, 42, 47, 52, 61]. To drive down communication and data structure overheads, other work only strives to maintain what we call *local ordering*. Threads push and pop to thread-private task queues and occasionally synchronize

*SPAA '24, June 17–21, 2024, Nantes, France*

with a global scheduler [31, 36, 56, 58]. These distributed designs alleviate some synchronization overhead and allow for a lightweight and scalable design, but local ordering makes threads more likely to work on lower-priority tasks and to wrongly execute *moot* [39] tasks. This improved scalability comes at the cost of redundant work [25, 40, 47]. In short, prior schedulers trade off idling threads, redundant work, and data structure inefficiency.

This paper presents the *Multi Bucket Queue* (MBQ),[1] a concurrent priority scheduler that targets an unexplored point in this design space (Sec. 2) to reap performance and efficiency improvements. The Multi Bucket Queue keeps threads busy with an asynchronous design, avoids useless work with global ordering, yet employs efficient underlying data structures. In short, our design marries the MultiQueue (MQ) [42] and the multi-level bucket queue [15, 17, 60], yet exploits their union with key optimizations (Sec. 3). We evaluate the Multi Bucket Queue across 8 applications relative to the MultiQueue [42], OBIM [36], PMOD [58], Julienne [17], and synchronous PBBS implementations [8] (Sec. 4). The Multi Bucket Queue achieves gmean speedup over the baseline MultiQueue by 5.3×; Julienne by 2.3×; PBBS by 1.6×; and is competitive with OBIM & PMOD with 1.07× and 1.01× slowdowns, respectively.

In short, the key contributions of this paper are:

- A taxonomy of existing concurrent priority schedulers.
- A characterization of *(i)* tradeoffs between work efficiency and performance across synchronous and asynchronous schedulers and *(ii)* scheduling overheads imposed by data structures.
- The MBQ, a previously unexplored point in the taxonomy.

## 2 MOTIVATION

Algorithms for the single-source shortest paths (sssp) problem on graphs with non-negative weights benefit from a dynamic priority task order and illustrate the tradeoffs in this work. Listing 1 shows Dijkstra's sequential algorithm that initializes the priority queue scheduler with the source vertex (line 5) then repeatedly pops the vertex with current minimum distance from the source. Each task (outer loop iteration) examines the prospective distances for v's neighbors and, upon finding a lower-distance path, re-inserts the neighbor into the priority queue as a new task (lines 12-15). Every vertex has a *rank* relative to all other vertices in the scheduler. Vertices with equal distance to the source are arbitrarily ordered and are viewed in the same *priority level*. Dijkstra's sequential algorithm is optimal for graphs with non-negative weights as it follows a strict priority order by popping the top-ranked vertex.

Unfortunately, this optimal sequential algorithm parallelizes poorly [31]. With a large number of threads, a scheduler must balance parallelism with the amount of redundant work generated. Many tasks pushed by Dijkstra's algorithm are *empty* [31] or *moot* [39]: they exit at line 9 without doing any work. All tasks that execute fully in the sequential implementation are *useful* work [31], as they set the vertex's distance to its final value. However, when moot tasks are popped out of order they could execute fully instead of exiting early, doing work that is redundant—does not contribute to the algorithm's output. Any descendants of these redundant tasks are themselves either moot or redundant, allowing wrongfully executed tasks to degrade the *work efficiency* [10] of the algorithm—the amount of work performed by the parallel algorithm compared to the best sequential one [17].

[1]We release the source code here https://github.com/mcj-group/mbq.

```
1 PriorityQueue pq;
2 int prios[G.n]; // current min priority per vertex
3 for (int v : G.V) prios[v] = INF;
4 prios[source] = 0;
5 pq.push(0, source); // start with source
6 while (!pq.empty()) {
7   int prio, v = pq.pop();
8   // early exit for moot tasks
9   if (prio > prios[v]) continue;
10  for (int nbr : G.edges[v]) {
11    int p = prio + distance(v, nbr);
12    if (p < prios[nbr]) {
13      prios[nbr] = p;
14      pq.push(p, nbr);
15    }
16  }
17 }
```

**Listing 1: The Dijkstra sssp algorithm using a priority queue.**

Prior software techniques and scheduler designs strive to scale parallel priority-ordered algorithm implementations with greater numbers of cores. However, these designs continue to face challenges in keeping *(i)* all cores well utilized, *(ii)* the algorithm work–efficient, and *(iii)* the implementation overheads in check. Pushing the scalability and efficiency of concurrent priority scheduling in software remains an active research problem.

### 2.1 The design space of priority schedulers

We present a taxonomy to understand these tradeoffs and how prior work navigates them (Sec. 2.2). We break the design space into three dimensions: *the degree of synchrony*, *the drift of priorities*, and *the underlying data structure*.

**The degree of synchrony** trades work efficiency for parallelism. Synchronous scheduling grants better work efficiency than asynchronous, but can be penalized with limited parallelism per superstep, leading to idling threads. Although asynchronous scheduling sidesteps the latter, it risks generating more redundant work.

A synchronous scheduler groups tasks into different priority levels separated by barriers. Tasks in the same priority level execute concurrently but all tasks in one level must complete before those in the next level can start. This maintains a strict priority order but restricts the amount of parallelism. With few tasks per priority level, many core cycles are wasted as threads spin at the barrier. With variable task length, synchronous scheduling is subject to one or more straggler threads that process the last few large tasks. Synchronous schedulers can *coarsen* [25] priorities to relax the order. Coarsening assigns tasks with similar priorities to the same priority level so the scheduler executes them as though they have equal priority. Coarsening boosts parallelism but can hurt work efficiency. Even the best performing coarsening factor can leave limited parallelism per priority level (Sec. 2.2).

Asynchronous schedulers provide flexibility as threads are not restricted by barriers. Asynchronous scheduling can be *speculative* for a strict priority order or *relaxed*. Although software-based speculation can reduce overheads by exploiting application-specific semantics like commutativity [9, 29], general-purpose speculation leads to intolerable overheads [12, 25, 26], making strict synchronous or relaxed asynchronous scheduling the only feasible choices

for most software. We do not consider speculative schedulers further. Relaxed asynchronous schedulers keep threads well utilized in part because they can pop and execute tasks out of priority order. Controlling the amount of generated redundant work becomes a key challenge. Most asynchronous schedulers provide no guarantees on the rank of the popped task [50]. With an unbounded rank for popped tasks, redundant work may become significant, hurting work efficiency and overall performance as a result.

**The drift of priorities: global order vs. local order.** Priority drift [46] trades communication cost for work efficiency. The closer a scheduler tracks the strict priority order, the better the work efficiency of its encapsulating algorithm. Unfortunately, limiting drift from the strict priority order can impose high communication overhead. Concretely, priority drift at a given time measures the average difference of priority between the globally most-prioritized task and the priority of tasks currently executing on all threads. Lower drift is a proxy for better work efficiency.

A scheduler that strives for *global ordering* has threads working on high priority (highly ranked) tasks. Unfortunately, threads frequently communicate with the scheduler to avoid drifting, imposing non-trivial synchronization overhead. An extreme example is a single lock-protected priority queue that is accessed by multiple threads. Although this scheduler concedes little priority drift, threads are blocked on each push and pop, hindering scalability.

A scheduler that allows for *local ordering* processes tasks in a low-communication, distributed manner. Typically, each thread has a private batch of tasks to process, and subsequently processes its own newly created tasks. With such a design, threads only occasionally synchronize with global scheduling structures. Although locally ordered systems reduce synchronization overhead, they are more likely to execute moot tasks and moot descendants, doing useless work that hurts their work efficiency.

**The underlying data structure** trades the sequential cost of scheduler operations for work efficiency. Scheduler overheads can significantly impact overall performance [31, 58] and much of that overhead is attributed to the underlying scheduling data structures [51] (Sec. 2.2). Asymptotically, a bucket pop takes $O(1)$ time, removing the tail from the underlying buffer, whereas a heap pop takes $O(\log n)$ time, growing with the $n$ queued tasks. Moreover, this high-level view hides the large constant factors caused by the memory hierarchy: buckets exhibit better locality than heaps [58]. For algorithms working on large heaps, such operation overhead becomes a burden to the scheduler. Similarly, designs built on sophisticated data structures are often bottlenecked on scheduler overhead [7, 20, 58]. On the other hand, an individual heap guarantees a strict priority order, whereas bucket queues with coarsened priority levels are prone to reordering. A more relaxed underlying data structure can hurt work efficiency. To improve overall performance, this dimension requires balancing an efficient and lightweight data structure with a modest increase in work.

## 2.2 Limitations of existing schedulers

Table 1 positions existing schedulers in our taxonomy. We characterize their tradeoffs below. Sec. 4.1 details our methodology.

**Synchronous schedulers can have limited work per barrier:** Julienne [17] is a synchronous, globally ordered framework that uses lightweight buckets as its data structure. Each bucket is an unordered buffer of tasks. Accessing tasks within a bucket exhibits spatial locality, allowing for high queue operation throughput. Despite this advantage, Julienne forgoes much parallelism.

**Table 1: Comparison of the MBQ and prior schedulers.**

| Design | Barrier-Free | Globally Ordered | Lightweight Structure |
|---|---|---|---|
| Julienne [17] | | ✔ | ✔ |
| Ord. GraphIt [60] | | ✔ | ✔ |
| MultiQueue [40] | ✔ | ✔ | |
| RELD [27] | ✔ | 🔗 | |
| Stealing MQ [42] | ✔ | ✔ | |
| SprayList [7] | ✔ | ✔ | |
| ZMSQ [61] | ✔ | ✔ | |
| k-LSM [56] | ✔ | | |
| OBIM [36] | ✔ | | ✔ |
| PMOD [36] | ✔ | | ✔ |
| **MBQ** (this paper) | ✔ | ✔ | ✔ |

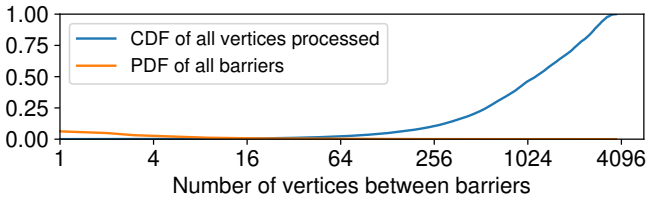


**Figure 1: Julienne's work distribution with Δ-stepping on USA roads using the Δ with the best 48-thread performance. Blue is the CDF of all vertices processed (y-axis) with increasing vertices per barrier (x-axis). Orange is the PDF of all barriers (y-axis) with a given number of vertices (x-axis).**

For example, Dijkstra's strict ordered algorithm on road graphs will often have fewer available tasks per priority level than there are threads. Even coarsening priorities with Δ-stepping [34] does not eliminate the problem. Fig. 1 shows the work distribution of Julienne's Δ-stepping on USA roads. The right side of the blue CDF shows that ~half of work is done in buckets with ≤ 1024 vertices. Even worse, the left side of the orange PDF shows that 10% of buckets hold only 1 vertex. Table 2 shows the average number of vertices/tasks per bucket and that the problem generalizes to other road graphs. With USA roads, all buckets have fewer than 4096 vertices. Even with a higher average degree on Europe roads, Julienne must face a large number of small buckets with limited parallelism per barrier. Fig. 2 translates these measurements to run time behavior: Julienne threads spend gmeans of 56% (`sssp`) and 26% (`sc`) of total execution time idling or sleeping at barriers. Given limited parallelism, Julienne fails to scale with core count.

Ordered GraphIt [60] addresses this issue by fusing consecutive buckets together to reduce the number of barriers. Although this optimization improves performance, the issue is not fully resolved as it remains synchronous: threads cannot execute tasks past the current bucket, preventing them from exploiting more parallelism.

**Table 2: Parallelism of Julienne's Δ-stepping on road graphs using the best-performing Δ for each graph. % Small Bkts is the fraction of buckets with fewer than 4096 tasks. Tasks/Bkt is the average number of tasks per barrier.**

| Input | #Vertices | #Edges | % Small Bkts | Tasks/Bkt |
|---|---|---|---|---|
| USA | 24 M | 58 M | 100% | 352 |
| USA-E | 4 M | 9 M | 100% | 716 |
| EUR | 57 M | 148 M | 60% | 7039 |
| NA | 33 M | 83 M | 66% | 4826 |

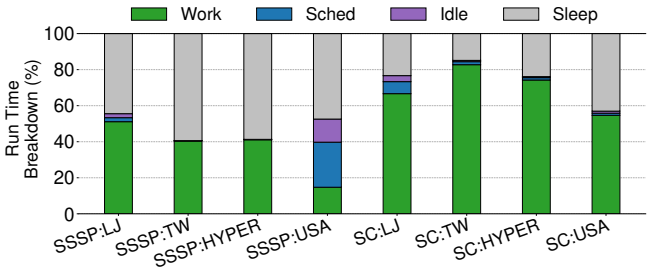


**Figure 2: Julienne run time breakdown at 48 threads for sssp and sc on various inputs.**

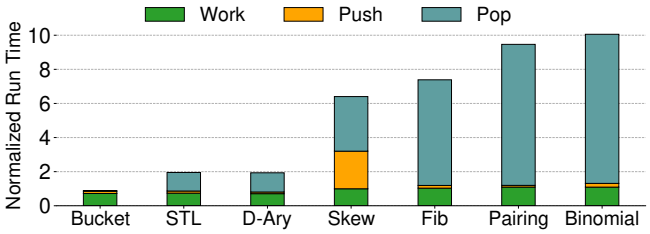


**Figure 3: Sequential run time of Dijkstra's algorithm on LJ using several heaps, normalized to the bucket queue. Queue overhead is the sum of push and pop time.**

**Async. schedulers often suffer data structure overhead:** The MultiQueue [42] is an asynchronous, globally ordered scheduler that uses heaps as the underlying priority queue (Table 1). It wraps $k$ lock-protected queues. For each pop, the rank of the returned task is $\leq O(k \log k)$ with high probability, and $O(k)$ on average [6]. With rank guarantees, the executed tasks closely follow the global priority order and generate little redundant work [40, 55].

Unfortunately, the MultiQueue's work efficiency is often overshadowed by data structure overhead. The $O(\log n)$ heapify operations (Sec. 2.1) are high compared to the short tasks of many encapsulating algorithms (e.g., Listing 1), and far more expensive than bucket operations. Fig. 3 shows the run-time breakdown of Dijkstra's sequential algorithm using different priority queue implementations [21, 23, 49, 52], normalized to the bucket queue (Sec. 3). These results corroborate prior work [51]: queue overheads dominate execution time. In the worst case, > 80% of run time is spent on pushes and pops with the Binomial heap. The Fibonacci heap provides the best asymptotic complexity in all operations, yet it is > 3× slower than a simpler 4-ary heap. Despite theoretically efficient heaps being developed over many years, these do not necessarily yield practical performance. This presents an opportunity for optimization on the MultiQueue.

RELD (remote enqueue, local dequeue) [27], the Stealing MultiQueue [40], and the SprayList [7] share design elements with the MultiQueue (Table 1). RELD is designed for hardware task parallelism where cores always pop from a local queue but push to random queues. The Stealing MultiQueue uses local heaps combined with random stealing to ensure a work-efficient schedule. The SprayList uses the SkipList [20, 41] as its underlying data structure, which also has average push/pop time complexity of $O(\log n)$. Like the MultiQueue, they all suffer from high overhead due to the underlying data structure [58]. The ZMSQ [61] builds on the Mound [32], which is similar to a heap but optimized for concurrent accesses. Mounds enable $O(\log \log n)$ pushes and $O(\log n)$ pops. The ZMSQ provides stronger rank guarantees and simpler termination than distributed CPS designs, allowing it to outperform the SprayList at low thread counts. However, it depends on a complex data structure and loses its advantage with more threads.

**Locally ordered schedulers increase priority drift:** OBIM [36] is an asynchronous, locally ordered scheduler with an efficient underlying data structure. OBIM groups tasks into priority levels (bags) and tracks the mappings of bags with a global metadata map. Each core maintains one local buffer per bag and a local map that caches the global map. A thread pushes a task to the local buffer corresponding to the task's priority level, and consumes its current buffer until it is empty, then synchronizes with the global map. As such, OBIM reduces communication overhead as threads mostly work on their local buffers. This approach trades off low scheduler overhead for a more relaxed order, and OBIM achieves superior performance compared to a plain MultiQueue. However, threads may be stuck working on tasks in the same bag for a long period of time without synchronizing with the global map, resulting priority inversion. PMOD [58] addresses this issue with an adaptive heuristic that dynamically changes the distribution of priority levels based on bag utilizations. Nevertheless, OBIM & PMOD's distributed design inevitably allows priority inversions and the schedule drifts away from the sequential order.

Fig. 4 breaks down the work done when executing `sssp` by the number of tasks executed. We observe that on USA roads, the number of tasks that execute redundant work (see beginning of Sec. 2) accounts for 38% of all tasks which do work. There are few moot tasks recorded on USA roads despite the high level of redundant work. This indicates that priority inversions allow some threads to waste significant time exploring suboptimal paths locally

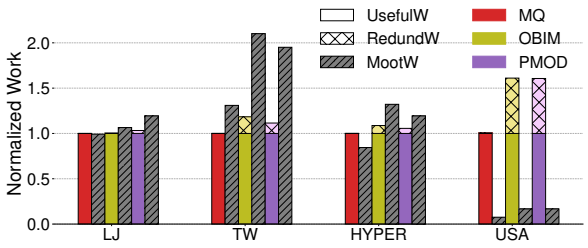


**Figure 4: Work normalized to the amount of useful work on sssp at 48 threads.**

before the global scheduler correct them. On social network graphs, OBIM & PMOD generate a large number of moot tasks, up to 2.1× the number of useful work. This abundance of short moot tasks cause significant overhead in performance (Sec. 4). The MultiQueue (shown as MQ) performs almost no redundant work and generates a small number of moot tasks, demonstrating that a globally ordered scheduler avoids the redundant work of locally ordered schedulers.

The k-LSM [56] similarly avoids contention by only occasionally synchronizing thread-local structures with a global queue. This approach suffers increasing priority drift as the number of threads increases, because its rank guarantee degrades linearly with both thread count and local queue capacity to account for the inaccessibility of the local queues of all other threads.

RELD occupies a middle ground where its pops from local queues degrade work efficiency compared to the MultiQueue [42], but its random pushes remain closer to a global ordering than a locally ordered CPS like OBIM or PMOD.

**Putting it all together:** We propose that a high-performance CPS should be *asynchronous* to allow for high scalability even on low-connectivity graphs such as road networks. It should maintain a *global ordering* to achieve work efficiency. Finally, it should use an *efficient* underlying data structure to reduce scheduler overhead.

## 3 THE MULTI BUCKET QUEUE

We design the Multi Bucket Queue (MBQ) to drive down thread idling, priority drift, and queue overheads. We achieve these goals by *(i)* allowing relaxation and using several lock-protected internal queues, *(ii)* having threads select random internal queues for pushes and pops, and *(iii)* using internal bucket queues. The first two draw on the strengths of the MultiQueue while the latter draws on the strength of Julienne and OBIM. We present the MBQ design in layers of abstraction and detail: the interface for threads and queue selection (Sec. 3.1), the bucket queue design (Sec. 3.2), and finally optimizations (Sec. 3.3). Without loss of generality, we assume the highest ranked task has minimum priority value.

### 3.1 Queue selection

Like the MultiQueue, the MBQ wraps $C \cdot T$ lock-protected internal queues, where $T$ is the number of threads and $C$ is a constant (4 in this paper). Listing 2 shows a simplified implementation. The push method selects a random uncontended queue as the destination. The pop method selects two random queues and removes the element from the queue with highest priority, if uncontended. pop may find that both queues are empty, but this does not imply that all queues are empty. Prior work details termination/emptiness detection [55] and we use their sssp approach for all algorithms.

### 3.2 Multi-level bucket queue

In place of a heap, the MBQ uses a multi-level bucket queue for the Queue type in Listing 2. Listing 3 shows the simplified implementation for one queue. A bucket queue maps every application priority level (e.g., vertex distance in sssp) to one bucket. We first assume support for an infinite number of priority levels (buckets), then handle finite storage.

Each Bucket is a resizable FIFO ring buffer of elements of type E. It has head and tail pointers. It initially allocates a starting capacity (default to 128) to minimally satisify the number of tasks in this priority level. The push method copies the input task into the tail location and increments tail with wraparound. When the Bucket is full, our implementation doubles the capacity[2] of the buffer with realloc. The pop method returns the task at head and increments head with wraparound, unless the Bucket is empty. These operations are constant time (amortized for push) and exhibit spatial locality for repeated calls. We exploit this opportunity with *task batching* (Sec. 3.3).

```
1 class MultiQueue<Queue, E> {
2   Queue<E> queues[C * T];
3
4   void push(E task) {
5     int q = random(0, C * T);
6     while (!queues[q].tryLock())
7       q = random(0, C * T);
8     queues[q].push(task);
9     queues[q].unlock();
10  }
11
12  E pop() {
13    while (true) {
14      int q1, q2 = randomTwo(0, C * T);
15      E top1 = queues[q1].top();
16      E top2 = queues[q2].top();
17      if (top1 < top2) swap(q1, q2);
18      if (!queues[q1].tryLock()) continue;
19      E task = queues[q1].pop();
20      queues[q1].unlock();
21      return task;
22    }
23  }
24 }
```

**Listing 2: Simplified MultiQueue implementation with templated Queue type. A heap Queue gives the classic MQ while a multi-level bucket queue gives a base MBQ.**

Each BucketQueue has an ordered set of Buckets (line 2). Its push method determines the priority level for task (line 7) and pushes it into the appropriate Bucket (line 8). In the common case, pop returns a task from the highest priority bucket (lines 16-17), tracked by bucket index minBkt. When that bucket is empty, minBkt advances by one to try popping from the next bucket (line 18). Conversely, if push receives a higher priority task than all queued tasks, minBkt drops to the incoming task's bucket (line 9).

A BucketQueue of infinite buckets is impractical, so we constrain the set to finite $N$ buckets. eToBucket extracts a task's priority level and maps it to one of the $N$ buckets in the current range. Like Julienne, it maps any task with priority level above the range to a reserved OVERFLOW bucket. Tasks in this bucket are completely unordered, so should not be popped from it directly. Instead, when pop advances minBkt to reach OVERFLOW (line 14), all other buckets are necessarily empty, so it unpacks the OVERFLOW bucket (lines 22-30). This shifts the mapping of priority levels upward and remaps all overflowed tasks to the new range. Unlike Julienne, the asynchrony of the MBQ allows for cases where push receives a task with priority level *below* the current range of this BucketQueue. eToBucket maps such tasks to the UNDERFLOW bucket. Such support for underflow is also required for algorithms with non-monotonically

[2] Bucket can downsize if underutilized, but we omit this: performance degrades due to *(i)* packing tasks to the start of buffer, and *(ii)* reallocations from fluctuating capacities.

```
class BucketQueue<E> {
  Bucket<E> buckets[N];
  int minBkt = INFINITY;
  int curRange; // The current range of mappings

  void push(E task) {
    int b = eToBucket(task);
    buckets[b].push(task);
    minBkt = MIN(minBkt, b);
  }

  E pop() {
    while (true) {
      if (minBkt == OVERFLOW) unpackOverflow();
      if (minBkt == OVERFLOW) return EMPTY_TASK;
      if (!buckets[minBkt].empty())
        return buckets[minBkt].pop();
      minBkt++;
    }
  }

  void unpackOverflow() {
    curRange++; // shifts the eToBucket mapping
    while (!buckets[OVERFLOW].empty()) {
      E task = buckets[OVERFLOW].pop();
      int b = eToBucket(task);
      if (b == OVERFLOW) push(task);
      else buckets[b].push(task);
    }
  }
}
```

**Listing 3: Simplified bucket queue implementation.**

increasing priorities. The `UNDERFLOW` constant is equal to zero to ensure it represents the highest priority bucket. The `UNDERFLOW` bucket typically has low occupancy as tasks rarely map below the current range. In the rare case (`astar`) where underflow has high occupancy, the `BucketQueue` unpacks the `UNDERFLOW` bucket in a way similar to `unpackOverflow` (not shown).

## 3.3 Optimizations

**Coarsening** is a well known technique [34] for bucket-based schedulers that increases parallelism when synchronous [17] or reduces communication frequency when asynchronous [36]. Although the MBQ is asynchronous, like (synchronous) Julienne, coarsening increases the number of tasks that fall within the active range of buckets. Like prior work, we apply this technique to a subset of algorithms by adding a programmer-defined scaling factor $\Delta$ in `eToBucket` to right-shift priority levels. More priority coarsening (a larger $\Delta$) causes fewer expensive calls to `unpackOverflow` but more task reordering and worse work efficiency.

**Batching** is another known optimization in the context of Multi-Queue descendents [40, 55]. The key idea is to amortize the cost to access a remote queue over several task pushes or pops. Heap-based schedulers use batching to amortize synchronization, but still suffer from high queue operation complexity and random memory accesses with poor locality (Sec. 2.2). In contrast, the MBQ is uniquely positioned to not only amortize syncronization cost, but to further exploit spatial locality: tasks reside in contiguous chunks per bucket. With the MBQ, task batching transforms individual queue operations into efficient contiguous memory accesses.

Our MBQ implementation gives each thread two thread-local, fixed-size buffers to hold tasks for `push` and `pop`. During `pop`, while the random internal queue is locked, the MBQ extracts a batch of tasks from the queue into the local pop buffer. To constrain priority drift, the MBQ only extracts tasks from the highest priority bucket, even if the bucket has fewer tasks than the pop-batch size. The thread's subsequent `pop` calls pull tasks from its local pop buffer.

While running a task, a thread builds a batch of children tasks to eventually `push` into the MBQ. When the push buffer fills up, `push` inserts the tasks to one random internal queue in bulk. When a thread depletes its pop buffer, before it pops another batch from a queue in bulk, it empties its push buffer by pushing the tasks to a random queue (like prior work [40]). This design decision increases communication cost but reduces priority drift, as each thread makes their tasks available to pop by other threads. Instead of directly consuming from the local push buffer like OBIM,[3] threads ensure that tasks are popped from a highly prioritized bucket.

**Prefetching** builds on task batching. After filling the pop buffer, each thread prefetches the necessary data (priorities, edge data, etc) for the tasks in its batch before it begins processing the tasks. Even with task batching, processing tasks still involves random memory reads/writes which are likely to miss in the cache. Prefetching for each batch transforms these misses into cache hits.

# 4 EVALUATION

We evaluate the Multi Bucket Queue performance across eight benchmarks that require, or benefit from, priority scheduling. We then tease apart the impact of its optimizations, and characterize its sensitivity to $\Delta$, batch size, and number of buckets.

## 4.1 Methodology

**Hardware:** We run experiments on an x86_64 machine with 187 GB of DRAM and 2 sockets, each with an Intel Platinum 8260 Cascade Lake processor running at 2.4 GHz. Each socket has 24 cores, for a total of 48 threads (hyper-threading disabled).

**Benchmarks:** Table 3 summarizes the benchmarks we use to evaluate the Multi Bucket Queue and other systems. We use the given priority-ordered variants of `pr` and `rbp` which terminate when the algorithm reaches a convergence threshold. We use two versions of `mis`: deterministic reservations [9] as a baseline and a relaxed version [5] for priority schdulers. For Julienne's `sc`, priorities are computed as logarithmic set costs to bound the total number of buckets. We found that Julienne's source code trims the input graph during cache warm-up and measures timing on the trimmed graph. We removed this trimming in our evaluation. `astar` inputs a graph augmented with geographical data for a more directed exploration than `ppsp`, prioritizing tasks according to the current path distance plus the great-circle distance to the destination. We adapt Swarm [27]'s implementation of `astar` for evaluation.

For every benchmark, we embed all schedulers (and the MBQ) within the given framework to ensure the non-scheduler data structures remain unchanged. For example, `sc` is derived from Julienne/Ligra source code, `sssp` is derived from Galois, and `mis` is derived

[3] In OBIM, a thread pushes a task to the local buffer that corresponds to the task's priority level, it can also pop from that same local buffer if it corresponds to the most prioritized level observed by the thread's local map.

**Table 3: Benchmarks and their baseline frameworks and implementations.**

| Abbrv. | Name | Implementation | Description |
|---|---|---|---|
| sssp | Single-Source Shortest Path | Galois [36] Δ-step [34] | Finds shortest paths to all vertices from a source. |
| ppsp | Point-to-Point Shortest Path | Galois [36] Δ-step [34] | Finds the shortest path from a source to a destination. |
| bfs | Breadth-First Search | Galois [36] | Finds the shortest path with unit edge weights to all vertices from a source. |
| pr | PageRank | Galois [36] Push-Based [53] | Computes and ranks the importance of vertices on a graph. |
| mis | Maximal Independent Set | PBBSv2 [8] Greedy [5] | Finds a set of non-adjacent vertices where all excluded vertices are adjacent to an included one. |
| sc | Approximate Set Cover | Ligra [48] Greedy [28] | Finds sets that are $O(\log n)$-approximate of the optimal cover. |
| rbp | Residual Belief Propagation | Relaxed [4] | Statistical inference algorithm used for Markov Random Fields. |
| astar | A∗ path finding [24] | Swarm [27]/Chronos [2] | Finds the shortest path from a source to a destination using geographically directed search. |

**Table 4: Input graphs.**

| Abbrv. | Input | #Vertices | #Edges |
|---|---|---|---|
| LJ | Soc-LiveJournal [14] | 5 M | 69 M |
| TW | Twitter [30] | 42 M | 1468 M |
| HYPER | Hyperlink2012-hosts [33] | 102 M | 2043 M |
| USA | USA roads [1] | 24 M | 58 M |
| GER | Germany roads [37] [a] | 12 M | 32 M |
| | Ising [13] [b] | 1 M | 2 M |

[a] Contains latitude and longitude coordinate data.
[b] $n \times n$ grid, each edge represents a pair of messages.

from PBBSv2, so we embed the schedulers within those frameworks, respectively. Task prioritization remains the same across schedulers for a given benchmark. Furthermore, we have verified that the bit representation of task descriptors (e.g., vertex ID) matches that in OBIM chunks and in Julienne buckets. This gives us confidence that difference in performance come from the difference in schedulers.

We compile all benchmarks with `-O3` using `clang` (version 16) with OpenCilk [45] (version 2.1) for `mis` and `sc`, and `gcc` (version 9) for all other benchmarks. We use C++ STL `seq_cst` memory order.

**Inputs:** Table 4 shows social network graphs, real-world road graphs, and synthetic graphs used as inputs to our benchmarks. The social network graphs follow the power-law distribution [35, 59]. Most vertices have few neighbors while a few vertices have many neighbors. We set edge weights of non-road input graphs to be uniformly random in the range $[1, 255]$. We use LJ, TW, HYPER, and USA for all benchmarks except `rbp` and `astar`. We evaluate `rbp` on a $10^3 \times 10^3$ Ising grid. We evaluate `astar` on Germany roads, as it contains latitude and longitude coordinate data.

**Tuning and Measurements:** For benchmarks and schedulers that require tuning (e.g., OBIM), we begin with a search for the optimal Δ value for each (benchmark, input, thread count) configuration. Fixing Δ, we then tune other parameters such as OBIM's chunk size and MBQ's batch size to find the optimal set of parameters. We use the MultiQueue (MQ) as baseline and its optimized variant (with prefetching and task batching) for further comparison. For the MBQ, we use 64 buckets per bucket queue. We report run times with a 10% trimmed mean, and perform enough runs to achieve 95% confidence intervals ≤ 6% for runs less than 1 second and ≤ 3% for runs > 1 second.

## 4.2 Performance of the Multi Bucket Queue

Table 5 and Table 6 show 1- and 48-thread tuned run times (**1T** & **48T**); slowdowns relative to the best 48-thread run time (**SD**), and the self-relative speedups scaling from 1 to 48 threads (**SC**). The optimized versions of the MBQ and the MQ are labeled with **-O**. At 48 threads, across all configurations, the MBQ-O outperforms the baseline MultiQueue by gmean 5.3× and up to 31× (`pr`,TW) and the MQ-O by gmean 1.8× and up to 8.9× (`pr`,TW). On `sssp`, `ppsp`, `bfs`, and `pr`, MBQ-O performs comparably with OBIM and PMOD, with 1.07× and 1.01× gmean slowdowns respectively. On `sssp`, `mis`, and `sc`, MBQ-O achieves gmean speedups of 1.6× and 2.3× over PBBS and Julienne, respectively. Fig. 5 shows normalized work across benchmarks, demonstrating that MBQ-O performs little redundant work as it maintains global ordering just like the MQ.

**`sssp`, `ppsp` and `bfs`:** MBQ-O is competitive with or exceeds OBIM and PMOD on power-law graphs. However, USA shows a limitation of the MBQ: the Galois schedulers outperform it, even though it has better work efficiency. Granted, for `sssp` on USA, MBQ-O is 9.4× faster than Julienne at 48-threads, corroborating prior limit studies of Julienne [60]. USA has a small frontier with little work per bucket, even with ideal coarsening (Sec. 2.2). Under these conditions, redundant work is a smaller overhead than the communication costs of maintaining a global ordering [58] for a moderate core count. Prior work [40] has found that OBIM's work increase on USA starts to degrade performance at over 64 cores, which is larger than our evaluation system. We prototyped a locally-ordered version of the Multi Bucket Queue and found that it outperformed OBIM and PMOD on this input on our system. HD-CPS [47] introduced the idea of a controller that dynamically switches a CPS between local and global ordering, and we leave integrating such a controller into the Multi Bucket Queue for future work.

**`pr` & `rbp`:** The tasks in these benchmarks have floating point math, making them longer than others, yet the MQs still causes significant scheduler overhead as its queues are bloated with tasks. On `pr`, MBQ-O outperforms OBIM and PMOD on USA and TW, and remains competitive for the other inputs, up to 15% slower (HYPER).

**`mis`:** We compare with the PBBS `incrementalMIS` which uses deterministic reservations [9]. In each round, PBBS attempts to process all unprocessed vertices in parallel, commits those that precede all their neighbours, and saves unprocessed vertices for the next round. The relaxed-ordered MBQ and the MQs perform many fewer iterations than PBBS, as they process vertices with a priority order rather than synchronous parallel steps. With the help of our optimizations and a carefully ordered schedule, the MBQ-O outperform PBBS across all inputs.

**Table 5: Overall Performance. 1T and 48T show the run time (seconds) for 1 thread and 48 threads, respectively, bolding the lowest. -O for MBQ and MQ denote optimized versions. SD reports the slowdown compared to the best run 48T time. Lower values are better with a value of 1 being the fastest, in dark green. SC shows the self-relative speedup scaling from 1 to 48 threads; the higher the better.**

| | | LiveJournal | | | | Twitter | | | | Hyperlink2012-hosts | | | | USA-roads | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Scheduler | 1 T | 48 T | SD | SC | 1 T | 48 T | SD | SC | 1 T | 48 T | SD | SC | 1 T | 48 T | SD | SC |
| sssp | **MBQ-O** | **1.72** | .106 | 1.05 | 16.3 | **24.6** | **1.24** | 1 | 19.9 | 43.6 | **1.98** | 1 | 22.1 | **2.35** | .482 | 1.46 | 4.88 |
| | OBIM | 2.43 | **.101** | 1 | 24.0 | 42.7 | 1.69 | 1.36 | 25.3 | 61.3 | 2.00 | 1.01 | 30.7 | 3.50 | .335 | 1.01 | 10.4 |
| | PMOD | 2.58 | .113 | 1.12 | 22.9 | 44.5 | 1.75 | 1.41 | 25.4 | 65.3 | 1.99 | 1.01 | 32.7 | 4.19 | **.331** | 1 | 12.7 |
| | Julienne | 1.80 | .182 | 1.80 | 9.90 | 29.6 | 3.05 | 2.46 | 9.71 | **43.0** | 3.33 | 1.69 | 12.9 | 2.54 | 4.52 | 13.7 | 0.562 |
| | MQ-O | 3.91 | .163 | 1.61 | 24.0 | 53.7 | 2.21 | 1.78 | 24.3 | 82.8 | 3.58 | 1.81 | 23.1 | 5.32 | .581 | 1.76 | 9.16 |
| | MQ | 4.65 | .527 | 5.22 | 8.83 | 60.2 | 5.34 | 4.30 | 11.3 | 93.1 | 8.59 | 4.34 | 10.8 | 6.48 | 1.08 | 3.27 | 5.98 |
| ppsp | **MBQ-O** | **1.63** | .109 | 1.10 | 15.0 | **29.8** | **1.22** | 1 | 24.4 | **34.1** | **1.25** | 1 | 27.2 | **2.51** | .322 | 1.67 | 7.80 |
| | OBIM | 2.52 | **.099** | 1 | 25.5 | 41.6 | 1.76 | 1.44 | 23.6 | 41.2 | 1.28 | 1.02 | 32.2 | 2.83 | **.193** | 1 | 14.6 |
| | PMOD | 2.67 | .114 | 1.15 | 23.4 | 44.4 | 1.73 | 1.41 | 25.7 | 43.0 | 1.35 | 1.08 | 31.7 | 3.26 | .194 | 1.01 | 16.8 |
| | MQ-O | 3.79 | .161 | 1.63 | 23.6 | 57.6 | 2.01 | 1.65 | 28.7 | 56.4 | 1.90 | 1.52 | 29.6 | 3.91 | .491 | 2.54 | 7.97 |
| | MQ | 4.51 | .512 | 5.17 | 8.81 | 63.4 | 4.99 | 4.08 | 12.7 | 62.4 | 4.60 | 3.67 | 13.6 | 5.15 | .780 | 4.04 | 6.60 |
| bfs | **MBQ-O** | **.629** | .043 | 1.16 | 14.6 | **11.5** | **.540** | 1 | 21.4 | **17.8** | .960 | 1.04 | 18.5 | **2.48** | .375 | 2.09 | 6.60 |
| | OBIM | .910 | **.037** | 1 | 24.6 | 14.8 | .698 | 1.29 | 21.2 | 23.9 | **.921** | 1 | 26.0 | 3.19 | **.179** | 1 | 17.8 |
| | PMOD | 1.04 | .042 | 1.14 | 24.8 | 16.8 | .848 | 1.57 | 19.8 | 26.8 | 1.01 | 1.10 | 26.5 | 3.25 | .202 | 1.13 | 16.1 |
| | MQ-O | 1.38 | .061 | 1.65 | 22.6 | 20.2 | .926 | 1.71 | 21.8 | 39.5 | 1.75 | 1.90 | 22.6 | 4.65 | .450 | 2.51 | 10.3 |
| | MQ | 1.75 | .288 | 7.78 | 6.09 | 22.8 | 2.42 | 4.48 | 9.45 | 45.5 | 4.49 | 4.88 | 10.1 | 5.70 | 1.01 | 5.66 | 5.63 |
| pr | **MBQ-O** | **62.2** | 2.99 | 1.04 | 20.8 | **760** | **32.3** | 1 | 23.6 | **1115** | 49.6 | 1.15 | 22.5 | **171** | **7.44** | 1 | 23.0 |
| | OBIM | 63.4 | **2.86** | 1 | 22.2 | 849 | 34.1 | 1.06 | 24.9 | 1190 | 45.2 | 1.05 | 26.3 | 256 | 7.71 | 1.04 | 33.2 |
| | PMOD | 67.4 | 2.95 | 1.03 | 22.8 | 835 | 34.0 | 1.05 | 24.5 | 1194 | **43.1** | 1 | 27.7 | 201 | 8.05 | 1.08 | 25.0 |
| | MQ-O | 173 | 7.35 | 2.57 | 23.6 | 12915 | 287 | 8.89 | 45.0 | 7618 | 264 | 6.13 | 28.9 | 541 | 23.7 | 3.19 | 22.8 |
| | MQ | 186 | 16.1 | 5.64 | 11.5 | 14499 | 1001 | 31.0 | 14.5 | 9092 | 893 | 20.7 | 10.2 | 589 | 55.5 | 7.45 | 10.6 |
| mis | **MBQ-O** | **.330** | **.027** | 1 | 12.2 | **3.81** | **.196** | 1 | 19.4 | **10.8** | **.394** | 1 | 27.5 | **1.15** | **.071** | 1 | 16.3 |
| | PBBS | .355 | .050 | 1.85 | 7.10 | 6.65 | .246 | 1.26 | 27.0 | 13.2 | .547 | 1.39 | 24.1 | 1.16 | .160 | 2.25 | 7.27 |
| | MQ-O | .737 | **.027** | 1 | 27.3 | 8.59 | .280 | 1.43 | 30.7 | 21.9 | .658 | 1.67 | 33.2 | 3.56 | .142 | 2.00 | 25.1 |
| | MQ | .976 | .217 | 8.04 | 4.50 | 11.1 | 2.01 | 10.3 | 5.53 | 29.0 | 7.71 | 19.6 | 3.76 | 4.92 | 1.38 | 19.4 | 3.57 |
| sc | **MBQ-O** | **2.41** | **.128** | 1 | 18.8 | 87.6 | 4.12 | 1.28 | 21.3 | 172 | **4.73** | 1 | 36.3 | **4.60** | **.172** | 1 | 26.8 |
| | Julienne | 3.29 | .412 | 3.22 | 7.97 | **48.9** | **3.22** | 1 | 15.2 | **111** | 5.32 | 1.12 | 20.9 | 11.7 | .806 | 4.69 | 14.5 |
| | MQ-O | 4.32 | .164 | 1.28 | 26.3 | 80.1 | 7.07 | 2.20 | 11.3 | 142 | 5.31 | 1.12 | 26.8 | 12.0 | .393 | 2.28 | 30.5 |
| | MQ | 4.77 | .535 | 4.18 | 8.92 | 83.2 | 7.41 | 2.30 | 11.2 | 149 | 9.56 | 2.02 | 15.6 | 14.9 | 1.95 | 11.3 | 7.67 |

**Table 6: Performance for `rbp` on Ising and `astar` on GER.**

| | | 1 T | 48 T | SD | SC |
|---|---|---|---|---|---|
| rbp | **MBQ-O** | **242** | **6.22** | 1 | 38.9 |
| | MQ-O | 462 | 9.86 | 1.58 | 46.9 |
| | MQ | 499 | 14.1 | 2.26 | 35.5 |
| astar | **MBQ-O** | **2.24** | **.172** | 1 | 13.0 |
| | MQ-O | 6.29 | .243 | 1.41 | 25.9 |
| | MQ | 6.87 | .264 | 1.53 | 26.0 |

**Table 7: The average cover sizes at 48 T for SC.**

| Scheduler | LJ | TW | HYPER | USA |
|---|---|---|---|---|
| **MBQ-O** | 956862 | 3846446 | 10449388 | 10711057 |
| Julienne | 956256 | 3845055 | 10447779 | 10706927 |
| MQ-O | 957241 | 3854513 | 10506879 | 10703222 |
| MQ | 956015 | 3845631 | 10449950 | 10703485 |

**sc:** The MBQ-O outperforms Julienne on LJ, HYPER and USA as it is barrier-free. Similarly, the MQ-O delivers on-par or even better performance than Julienne on these graphs. However, on the highly connected TW input there is a large number of cardinality updates,

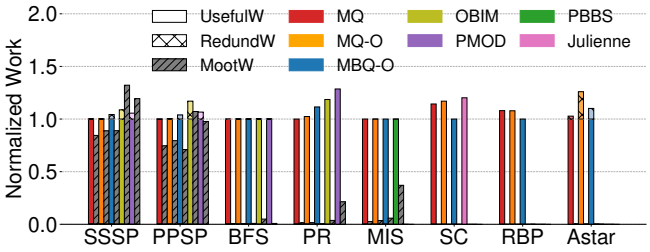


**Figure 5: Work normalized to the amount of useful work (`sssp`, `ppsp`, `bfs`, `mis` and `astar`) or the lowest amount of work (`pr`, `rbp` and `sc`). We use Ising as input for `rbp`, GER for `astar`, and HYPER for the other benchmarks.**

incurring significant overhead in re-bucketing sets. Prior work [60] found that Julienne's method of lazily updating the buckets is highly efficient, which matches our findings on TW.

We recognize that `sc` is a greedy algorithm. Unlike deterministic versions, MBQ's reordering of tasks can produce a lower quality output, resulting in a larger cover size compared to sequential execution. Table 7 shows that our relaxed implementation of `sc` achieves only slightly larger cover sizes compared to Julienne.

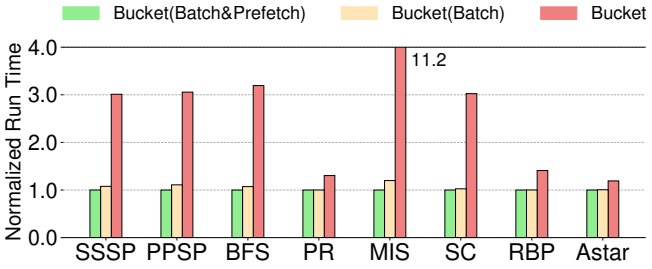


**Figure 6: The run times of different versions of the Multi Bucket Queue at 48 threads, normalized to MBQ-O (in green). Lower is better. The inputs are Ising for `rbp`, GER for `astar`, and HYPER for the rest.**

## 4.3 Impact of optimizations

We perform an ablation study to evaluate the performance improvement of our optimizations to the MBQ. Fig. 6 shows 48-thread run time on all benchmarks normalized to MBQ-O (in green) then incrementally disables prefetching then task batching. First, disabling prefetching (in yellow) increases run time by gmean 6% and up to 20% (`mis`). Next, disabling task batching (in red) results in serious performance degradation: relative to MBQ-O, run times are gmean 2.6× longer, up to 11.2× (`mis`). As described in Sec. 3.3, the structure of buckets naturally exhibits locality, and task batching fully exploits this opportunity.

## 4.4 Sensitivity studies

**Tuning Δ:** The Δ value is a programmer-defined parameter that affects the distribution of priority levels. Increasing Δ coarsens priorities, increasing the available parallelism within each priority level. With more tasks in a priority level, the Multi Bucket Queue can benefit more from locality, improving performance. However, the Δ value cannot be increased infinitely, as it also increases priority inversion. A maximally coarsened priority schedule is unordered, resulting in redundant work and worse work efficiency.

Fig. 7 shows the effect of changing Δ for `sssp`, `mis` and `rbp`. The blue bars show the 48-thread MBQ run times, and the red dots show the amount of work performed, both normalized to the Δ with the best run time. For `mis`, the best and worst performing Δ differ by 3× in run time. `rbp` shows a clear impact of work efficiency. When the Δ is small, the algorithm performs less work to converge. As Δ increases, the amount of work also increases. Once Δ passes

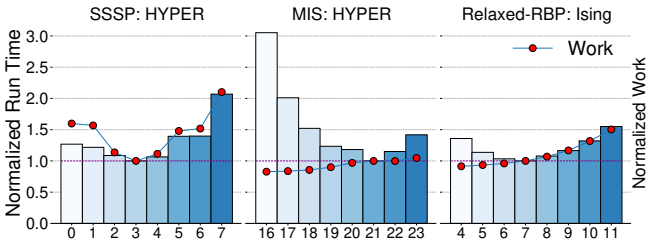


**Figure 7: Sweeping Δ values for the MBQ on `sssp`, `mis` and `rbp`. Δ is in powers of 2 and the x-axis shows the power. The y-axis is 48-thread run time (bars) and amount of work (dots), both normalized to the best performing Δ. Lower is better.**

the optimal point ($2^7$), performance degrades. Tuning the Δ value manages the tradeoff between work efficiency and performance.

**Batch sizes:** We evaluate performance with push & pop batch sizes ranging from 1 to 1024.[4] We extend the batch sizes enough to observe diminishing returns for most benchmarks and inputs. A larger batch size can reduce scheduler overhead, as more task pushes or pops amortize the queue lock acquisition. This reduces contention between threads and improves locality, at the cost of potentially higher priority drift. The batched pops are contiguous memory accesses; batched pushes can also benefit from locality when most tasks are mapped to the same priority level.

Fig. 8 shows that all configurations benefit from task batching and can achieve a speedup of up to 13× compared to no batching. `mis`, `sc`, and the search algorithms (on HYPER), benefit from large batches, as there are plenty of vertices in each priority level. For `mis`, since our implementation is priority-ordered, the MBQ and the MQs won't be penalized with many re-inserted moot tasks, even with a large batch size (see Fig. 5). This enables the use of aggressive batching for higher performance. For `sc`, majority of the popped tasks end up being moot and are immediately re-inserted back into the scheduler with an updated priority. This issue is even worse on highly connected inputs that generate a large number of cardinality updates. Therefore, high pop throughput filters out these tasks quickly, and a large push batch size greatly reduces the communication overhead.

On the other hand, smaller batch sizes make threads synchronize more frequently. Since the MBQ does not pop tasks from local push buffers, a lower drift improves performance when there is less parallelism per bucket (e.g., `bfs` on USA). Furthermore, the best configuration does not necessarily have the same batch size for push and pop. Future work could improve the MBQ by dynamically tuning the batch sizes.

**Number of buckets & priority levels:** Depending on the input and the benchmark, the distribution of tasks across priority levels may be skewed. For `sc` on power-law graphs, most buckets contain few vertices. Julienne's implementation of `sc` prioritizes sets using the logarithm of their costs to flatten this distribution. Without this, Julienne's `sc` at 48 threads is 22× and 9× slower on TW and HYPER respectively. The MBQ achieves the performance in Table 5 without this optimization. Since the MBQ is barrier-free and relaxed, threads can pop from different priority levels and do not need to wait. With the MBQ, threads quickly pass through small buckets even with a skewed distribution of priorities.

# 5 RELATED WORK

Many algorithms use priority scheduling to eliminate redundant work or for faster convergence [18, 19, 24, 28, 34, 43, 50]. Optimized sequential priority queue data structures, such as the Fibonacci heap [21], Binomial heap [52], and SkipList [20, 41], provide improved theoretical time complexity over binary heaps [54], but they do not perform as well in practice [51]. Synchronizing concurrent accesses to these priority queues using techniques such as flat-combining [22] and elimination [11] reduces contention, but they struggle to scale to high thread counts. This issue motivates

[4]These are maximum batch sizes, serving as limits to thread-local buffers (see Sec. 3.3). The buffers do not necessarily fill up.

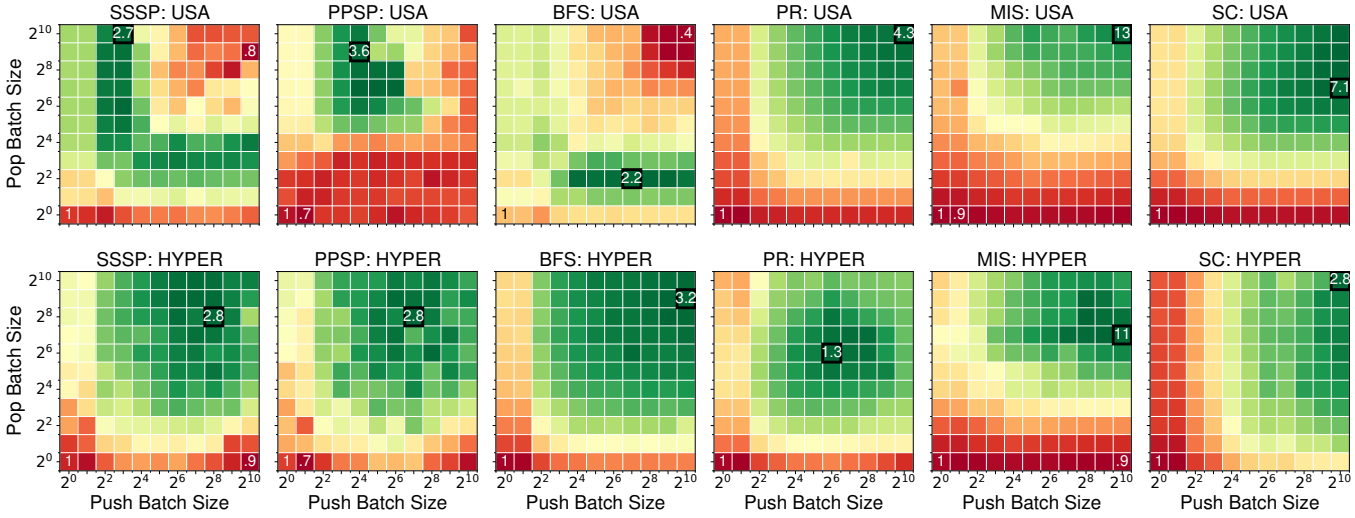


**Figure 8: Performance impact of push (x-axis) and pop (y-axis) batch sizes for the Multi Bucket Queue at 48 threads across various benchmarks on two vastly different graphs: USA and HYPER. Each cell represents speedup normalized to batch size of (1, 1) in the bottom left corner per plot. The best speedup in green is highlighted with a black box and white text. The worst performance in red is also labeled if $< 1$.**

relaxed concurrent priority schedulers that trade away the strict priority order for scalability.

With the proliferation of manycore architectures, parallel schedulers build atop efficient data structures to attempt to scale algorithms to higher thread counts. Afek et al. [3] explore the tradeoff of scalability versus strict linearizability. Wimmer et al. [57] apply relaxation to design work-stealing and $k$-priority data structures. The SprayList [7] and the lock-free $k$-LSM priority queue [56] are asynchronous schedulers that exploit relaxation. The MultiQueue [42] builds on the ideas of relaxation and bulk parallel heaps [16, 44]. Additional optimizations such as task batching [55] target the MultiQueue to amortize the cost of queue operations. The Stealing MultiQueue [40] adapts the MultiQueue to exploit locality and NUMA-awareness and improves performance over the MultiQueue. Each thread pops from a local queue, but uses work stealing to provide a superior rank guarantee over RELD. The Multi Bucket Queue continues this tradition by generalizing the asynchronous and scalable MultiQueue's structure beyond heaps.

Although designs like the MultiQueue constrain their relaxation with theoretical rank guarantees, others have explored designs for high scalability and low overheads. Lenharth et al. [31] showed that work efficiency is not the most important concern for performance, and the Galois OBIM scheduler [31, 36] optimizes for low scheduler overhead. With its distributed design and batching tasks into chunks, OBIM achieves high scalability and outperforms the SprayList and the MultiQueue. PMOD [58] improves upon OBIM with an adaptive heuristic that increases OBIM's bag utilization and reduces redundant work, all while reducing the need for tuning. The Multi Bucket Queue applies the efficiency of asynchronous OBIM/PMOD buckets to the global ordering of the MultiQueue.

Synchronous schedulers like Julienne [17], Ordered GraphIt [60], and Kinetic Dependence Graphs [26] use lightweight data structures to store tasks and can apply coarsening to increase parallelism. However, they perform poorly on graphs or algorithms with few tasks per priority level. The Multi Bucket Queue keeps the global ordering property of these schedulers to maintain work efficiency, while discarding their synchronous execution.

## 6 CONCLUSION

We have presented a taxonomy which classifies concurrent priority schedulers (CPS) according to their synchrony, drift, and underlying data structure. By classifying existing schedulers with our taxonomy we have analyzed their strengths and weaknesses, and identified an unexplored point in the design space. We have designed and implemented the Multi Bucket Queue, a CPS which combines the strengths of prior designs, and avoids their weaknesses. The Multi Bucket Queue is competitive with or exceeds the performance of state-of-the-art designs across a variety of applications and inputs.

## ACKNOWLEDGMENTS

We sincerely thank Aster Plotnik and the anonymous reviewers for their helpful feedback. This work was supported in part by the Digital Research Alliance of Canada, the University of Toronto, NSERC, an Ontario Queen Elizabeth II Graduate Scholarship, and an Ontario Bell Graduate Scholarship.

## REFERENCES

[1] 2006. 9th DIMACS Implementation Challenge: Shortest Paths. http://www.dis.uniroma1.it/~challenge9, archived at https://perma.cc/5KYT-YM36. (2006).

[2] Maleen Abeydeera and Daniel Sanchez. 2020. Chronos: efficient speculative parallelism for accelerators. In *Proc. ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. DOI: 10.1145/3373376.3378454.

[3] Yehuda Afek, Guy Korland, and Eitan Yanovsky. 2010. Quasi-Linearizability: Relaxed Consistency for Improved Concurrency. In *Proceedings of the International Conference on Principles of Distributed Systems*. DOI: 10.1007/978-3-642-17653-1_29.

[4] Vitaly Aksenov, Dan Alistarh, and Janne H. Korhonen. 2020. Relaxed scheduling for scalable belief propagation. In *Proc. of the International Conference on Neural Information Processing Systems*.

[5] Dan Alistarh, Trevor Brown, Justin Kopinsky, and Giorgi Nadiradze. 2018. Relaxed schedulers can efficiently parallelize iterative algorithms. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*. DOI: 10.1145/3212734.3212756.

[6] 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*. DOI: 10.1145/3087801.3087810.

[7] Dan Alistarh, Justin Kopinsky, Jerry Li, and Nir Shavit. 2015. The SprayList: A scalable relaxed priority queue. In *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. DOI: 10.1145/2688500.2688523.

[8] Daniel Anderson, Guy E. Blelloch, Laxman Dhulipala, Magdalen Dobson, and Yihan Sun. 2022. Brief announcement: the problem based benchmark suite. In *Proc. of the ACM SIGPLAN symp. on Principles and Practice of Parallel Programming*. DOI: 10.1145/3503221.3508422.

[9] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. 2012. Internally deterministic parallel algorithms can be fast. In *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. DOI: 10.1145/2145816.2145840.

[10] Guy E. Blelloch and Bruce M. Maggs. 1996. Parallel algorithms. *ACM Computing Surveys*, 28, 1, 51–54. DOI: 10.1145/234313.234339.

[11] Irina Calciu, Hammurabi Mendes, and Maurice Herlihy. 2014. The adaptive priority queue with elimination and combining. In *International Symposium on Distributed Computing*. DOI: 10.1007/978-3-662-45174-8_28.

[12] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. 2008. Software transactional memory: why is it only a research toy? *Queue*, 6, 5, 46–58. DOI: 10.1145/1454456.1454466.

[13] Barry A. Cipra. 1987. An introduction to the Ising model. *The American Mathematical Monthly*, 94, 10, 937–959. DOI: 10.1080/00029890.1987.12000742.

[14] Timothy A. Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*. DOI: 10.1145/2049662.2049663.

[15] Eric V. Denardo and Bennett L. Fox. 1979. Shortest-route methods: 1. reaching, pruning, and buckets. *Operations Research*, 27, 161–186, 1. DOI: 10.1287/opre.27.1.161.

[16] Narsingh Deo and Sushil Prasad. 1992. Parallel heap: an optimal parallel priority queue. *The Journal of Supercomputing*, 6. DOI: 10.1007/BF00128644.

[17] Laxman Dhulipala, Guy Blelloch, and Julian Shun. 2017. Julienne: a framework for parallel graph algorithms using work-efficient bucketing. In *Proc. ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. DOI: 10.1145/3087556.3087580.

[18] Edsger W. Dijkstra. 1959. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1, 1, 269–271. DOI: 10.1007/BF01386390.

[19] Gal Elidan, Ian McGraw, and Daphne Koller. 2006. Residual belief propagation: informed scheduling for asynchronous message passing. In *Proc. Conference on Uncertainty in Artificial Intelligence*.

[20] Keir Fraser. 2004. *Practical lock-freedom*. Ph.D. Dissertation. University of Cambridge. DOI: 10.48456/tr-579.

[21] Michael Fredman and Robert Tarjan. 1984. Fibonacci heaps and their uses in improved network optimization algorithms. In *Proc. IEEE Symposium on Foundations of Computer Science*. DOI: 10.1109/SFCS.1984.715934.

[22] Michael Fredman and Robert Tarjan. 2010. Flat combining and the synchronization-parallelism tradeoff. In *Proc. ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. DOI: 10.1145/1810479.1810540.

[23] Michael L. Fredman, Robert Sedgewick, Daniel D. Sleator, and Robert E. Tarjan. 1986. The pairing heap: a new form of self-adjusting heap. *Algorithmica*, 1, 111–129. DOI: 10.1007/BF01840439.

[24] Peter Hart, Nils Nilsson, and Bertram Raphael. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Systems Science and Cybernetics*, 4, 2. DOI: 10.1109/TSSC.1968.300136.

[25] Muhammad Amber Hassaan, Martin Burtscher, and Keshav Pingali. 2011. Ordered vs. unordered: A comparison of parallelism and work-efficiency in irregular algorithms. In *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. DOI: 10.1145/2038037.1941557.

[26] Muhammad Amber Hassaan, Donald Nguyen, and Keshav Pingali. 2015. Kinetic Dependence Graphs. In *Proc. ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. DOI: 10.1145/2775054.2694363.

[27] Mark C. Jeffrey, Suvinay Subramanian, Cong Yan, Joel Emer, and Daniel Sanchez. 2015. A scalable architecture for ordered parallelism. In *Proc. IEEE/ACM International Symposium on Microarchitecture (MICRO)*. DOI: 10.1145/2830772.2830777.

[28] David S. Johnson. 1974. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, 42. DOI: 10.1145/800125.804034.

[29] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. 2007. Optimistic parallelism requires abstractions. In *Proc. ACM-SIGPLAN Symposium on Programming Language Design and Implementation (PLDI)*. DOI: 10.1145/1250734.1250759.

[30] Haewoon Kwak, Changhyun Lee, and Hosung Park andSue Moon. 2010. What is Twitter, a social network or a news media? In *Proc. ACM International Conference on World Wide Web (WWW)*. DOI: 10.1145/1772690.1772751.

[31] Andrew Lenharth, Donald Nguyen, and Keshav Pingali. 2015. Priority queues are not good concurrent priority schedulers. In *Proc. European Conference on Parallel Processing (Euro-Par)*. DOI: 10.1007/978-3-662-48096-0_17.

[32] Yujie Liu and Michael Spear. 2012. Mounds: array-based concurrent priority queues. In *Proc. International Conference on Parallel Processing (ICPP)*, 1–10. DOI: 10.1109/ICPP.2012.42.

[33] Robert Meusel, Oliver Lehmberg, Christian Bizer, and Sebastiano Vigna. 2014. Hyperlink graphs. Web Data Commons. (2014). https://webdatacommons.org/hyperlinkgraph.

[34] U. Meyer and P. Sanders. 2003. Delta-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms*, 49, 1. DOI: 10.1016/S0196-6774(03)00076-2.

[35] Seth A. Myers, Aneesh Sharma, Pankaj Gupta, and Jimmy Lin. 2014. Information network or social network? the structure of the Twitter follow graph. In *Proc. ACM International Conference on World Wide Web (WWW)*. DOI: 10.1145/2567948.2576939.

[36] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A lightweight infrastructure for graph analytics. In *Proc. ACM Symposium on Operating Systems Principles (SOSP)*. DOI: 10.1145/2517349.2522739.

[37] OpenStreetMap. [n. d.] \NoCaseChange{http://www.openstreetmap.org}. ().

[38] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. 2011. The tao of parallelism in algorithms. In *Proc. ACM-SIGPLAN Symposium on Programming Language Design and Implementation (PLDI)*. DOI: 10.1145/1993498.1993501.

[39] Gilead Posluns, Yan Zhu, Guowei Zhang, and Mark C. Jeffrey. 2022. A scalable architecture for reprioritizing ordered parallelism. In *Proc. IEEE/ACM International Symposium on Computer Architecture (ISCA)*. DOI: 10.1145/3470496.3527387.

[40] Anastasiia Postnikova, Nikita Koval, Giorgi Nadiradze, and Dan Alistarh. 2022. Multi-queues can be state-of-the-art priority schedulers. In *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. DOI: 10.1145/3503221.3508432.

[41] William Pugh. 1990. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33, 6, 668–676. DOI: 10.1145/78973.78977.

[42] Hamza Rihani, Peter Sanders, and Roman Dementiev. 2015. Multiqueues: simple relaxed concurrent priority queues. In *Proc. ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. DOI: 10.1145/2755573.2755616.

[43] Peter Sanders. 2000. Fast priority queues for cached memory. *ACM Journal of Experimental Algorithmics (JEA)*, 5. DOI: 10.1145/351827.384249.

[44] Peter Sanders. 1998. Randomized Priority Queues for Fast Parallel Access. *Journal of Parallel and Distributed Computing*, 49. DOI: 10.1006/jpdc.1998.1429.

[45] Tao B. Schardl and I-Ting Angelina Lee. 2023. Opencilk: a modular and extensible software infrastructure for fast task-parallel code. In *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. DOI: 10.1145/3572848.3577509.

[46] Mohsin Shan and Omer Khan. 2021. Accelerating concurrent priority scheduling using adaptive in-hardware task distribution in multicores. *IEEE Computer Architecture Letters (CAL)*, 20, 1, 17–21. DOI: 10.1109/LCA.2020.3045670.

[47] Moshin Shan and Omer Khan. 2022. Hd-cps: hardware-assisted drift-aware concurrent priority scheduler for shared memory multicores. In *Proc. IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. DOI: 10.1109/HPCA53966.2022.00046.

[48] Julian Shun and Guy E Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. DOI: 10.1145/2442516.2442530.

[49] Daniel Dominic Sleator and Robert Endre Tarjan. 1986. Self-adjusting heaps. *SIAM Journal on Computing*, 15. DOI: 10.1137/0215004.

[50] Marvin Stümmel, Felix Brüll, and Thorsten Grosch. 2022. Relaxed parallel priority queue with filter levels for parallel mesh decimation. In *Proc. International Symposium on Vision, Modeling, and Visualization (VMV)*. DOI: 10.2312/vmv.20221202.

[51] Ami Tavory, Vladimir Dreizin, and Benjamin Kosnik. 2004. Priority-queue performance tests. In *Policy-Based Data Structures*. GNU Operating System, (Feb. 2004). Chap. 5.2.2. https://gcc.gnu.org/onlinedocs/libstdc++/ext/pb_ds/pq_performance_tests.html.

[52] Jean Vuillemin. 1978. A data structure for manipulating priority queues. *Communications of the ACM*, 21. DOI: 10.1145/359460.359478.

[53] Joyce Jiyoung Whang, Andrew Lenharth, Inderjit S. Dhillon, and Keshav Pingali. 2015. Scalable data-driven pagerank: algorithms, system issues, and lessons learned. In *Proc. European Conference on Parallel Processing (Euro-Par)*. DOI: 10.1007/978-3-662-48096-0_34.

[54] J. W. J Williams. 1964. Algorithm 232 Heapsort. *Communications of the ACM*, 7, 6, 347–349. DOI: 10.1145/512274.512284.

[55] Marvin Williams, Peter Sanders, and Roman Dementiev. 2021. Engineering multiqueues: fast relaxed concurrent priority queues. In *Proc. European Symposium on Algorithms (ESA)*. DOI: 10.4230/LIPIcs.ESA.2021.81.

[56] Martin Wimmer, Jakob Gruber, Jesper Larsson Träff, and Philippas Tsigas. 2015. The lock-free k-LSM relaxed priority queue. In *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. DOI: 10.1145/2858788.2688547.

[57] Martin Wimmer, Francesco Versaci, Jesper Träff, Daniel Cederman, and Philippas Tsigas. 2014. Data structures for task-based priority scheduling. In *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. DOI: 10.1145/2692916.2555278.

[58] Serif Yesil, Azin Heidarshenas, Adam Morrison, and Josep Torrellas. 2019. Understanding priority-based scheduling of graph algorithms on a shared-memory platform. In *Proc. ACM/IEEE Supercomputing Conference (SC)*. DOI: 10.1145/3295500.3356160.

[59] Pavel Zakharov. 2007. Diffusion approach for community discovering within the complex networks: livejournal study. *Physica A: Statistical Mechanics and its Applications*. DOI: 10.1016/j.physa.2006.11.086.

[60] Yunming Zhang, Ajay Brahmakshatriya, Xinyi Chen, Laxman Dhulipala, Shoaib Kamil, Saman Amarasinghe, and Julian Shun. 2020. Optimizing ordered graph algorithms with GraphIt. In *Proc. of the ACM/IEEE International Symposium on Code Generation and Optimization (CGO)*. DOI: 10.1145/3368826.3377909.

[61] Tingzhe Zhou, Maged Michael, and Michael Spear. 2019. A practical, scalable, relaxed priority queue. In *Proceedings of the 48th International Conference on Parallel Processing*. DOI: 10.1145/3337821.3337911.