

---

# MULTI-QUEUES CAN BE STATE-OF-THE-ART PRIORITY SCHEDULERS

---

**Anastasiia Postnikova**

ITMO University

postnikovaanastasiaa@gmail.com

**Nikita Koval**

JetBrains Research

nikita.koval@jetbrains.com

**Giorgi Nadiradze**

IST Austria

giorgi.nadiradze@ist.ac.at

**Dan Alistarh**

IST Austria

dan.alistarh@ist.ac.at

September 3, 2021

## ABSTRACT

Designing and implementing efficient parallel priority schedulers is an active research area. An intriguing proposed design is the *Multi-Queue*: given  $n$  threads and  $m \geq n$  distinct priority queues, task insertions are performed uniformly at random, while, to delete, a thread picks two queues uniformly at random, and removes the observed task of higher priority. This approach scales well, and has probabilistic rank guarantees: roughly, the rank of each task removed, relative to remaining tasks in all other queues, is  $O(m)$  in expectation. Yet, the performance of this pattern is below that of well-engineered schedulers, which eschew theoretical guarantees for practical efficiency.

We investigate whether it is possible to design and implement a Multi-Queue-based task scheduler that is both highly-efficient and has analytical guarantees. We propose a new variant called the *Stealing Multi-Queue (SMQ)*, a *cache-efficient* variant of the Multi-Queue, which leverages both *queue affinity*—each thread has a *local* queue, from which tasks are usually removed; but, with some probability, threads also attempt to *steal* higher-priority tasks from the other queues—and *task batching*, that is, the processing of several tasks in a single insert / remove step. These ideas are well-known for task scheduling *without priorities*; our theoretical contribution is showing that, despite relaxations, this design can still provide rank guarantees, which in turn implies bounds on total work performed. We provide a general *SMQ* implementation which can surpass state-of-the-art schedulers such as Galois and PMOD in terms of performance on popular graph-processing benchmarks. Notably, the performance improvement comes mainly from the *superior rank guarantees* provided by our scheduler, confirming that analytically-reasoned approaches can still provide performance improvements for priority task scheduling.

## 1 Introduction

Scalable concurrent priority schedulers are a key ingredient for efficiently parallelizing algorithms arising in graph processing, computational geometry, or scientific simulation. In such settings, algorithms are usually structured as a series of prioritized tasks, each of which accesses a subset of the shared algorithm state, computes a local update, and applies the update to the shared state. Many classic problems fit this pattern, such as graph processing algorithms (e.g., BFS or MST computations), Dijkstra’s single-source shortest paths (SSSP), or Delaunay mesh triangulation.

Often, algorithm semantics imply a natural task priority ordering. A classic example is Dijkstra’s SSSP algorithm, where the first task to be processed should be the one corresponding to the active node that is closest to the source. Concurrent schedulers often relax this sequential order to enable parallelism, but also, importantly, to reduce the overhead of the task scheduler’s implementation, which can become extremely contended if perfect priority order is

enforced [17]. However, relaxing the priority order excessively can also decrease performance, as it leads to wasted work: in Dijkstra’s SSSP algorithm, for example, processing a node out-of-order at a distance that is higher than its minimal distance from the source is useless, since the node will need to be re-processed later, when its distance is updated to the correct one. Generally, relaxed priority schedulers induce non-trivial trade-offs between the higher scalability of the scheduling mechanism itself, allowed by out-of-order execution, and the potential for wasted work caused by excessive speculation.

The sustained recent progress on concurrent priority scheduling can be viewed through the lens of this trade-off. The influential Galois line of work, e.g. [19, 22] proposed a family of highly-efficient schedulers which significantly relax priority order, specifically focusing on performance rather than priority guarantees [17]. As a consequence, the scheduler has low overhead, but may induce high wasted work [27]. This approach provided state-of-the-art performance upon its publication, and has inspired significant follow-up work, notably in terms of hardware implementations, e.g. [1, 15]. Yesil et al. [27] performed an in-depth analysis of the relaxation-vs-wasted-work trade-off in concurrent priority schedulers, and proposed a new scheduler called PMOD, combining the high scalability of Galois-type schedulers, with a dynamic priority management heuristic which reduces wasted work.

A parallel research thread has been on providing efficient relaxed schedulers *with guarantees* on the maximum amount of priority rank relaxation, e.g. [6, 23, 26, 24]. Of these, arguably the most popular design is the Multi-Queue [23], which works roughly as follows: given  $n$  threads, we instantiate  $m \geq n$  concurrent priority queues, which will store tasks. To *insert* a task, a thread simply places it into a random priority queue. To *remove* a task, the thread picks *two* priority queues at random, and removes the top element of higher priority. (For simplicity, we discuss concurrency-related details in the later sections.) Alistarh et al. [5] showed that a sequential variant of the Multi-Queue ensures that the rank of an element removed is always  $O(m)$ , in expectation, and  $O(m \log m)$ , with high probability in the number of queues  $m$ . Follow-up work showed that similar guarantees can be extended to concurrent executions [3], and gave work bounds for some task-based algorithms when executed via Multi-Queue-like priority schedulers [4, 7, 2]. Despite their guarantees, Multi-Queue schedulers are known to have lower overall performance relative to efficient scheduling heuristics [17, 27].

**Contribution.** In this paper, we show that these two lines of work can be unified, providing a highly-efficient, practical concurrent priority scheduler, while still maintaining theoretical rank guarantees, under analytical assumptions, for variants of the scheduler.

An obvious reason for the lower performance of Multi-Queues relative to practical heuristics is poor cache efficiency, as the basic process requires a high number of random accesses to maintain rank guarantees. A simple practical approach to address such issues, which we also adopt, is to affinitize threads to queues, assigning some subset of queues “preferentially” to each thread. For insertions, each thread can pick one of these preferential queues with (higher) probability  $p_{insert}$ , relative to inserting into other queues.

We would like to apply a similar approach for deletions. Yet, allowing fully-local removals would perturb the two-choice process, and cause divergence [21, 5]. Nevertheless, we show that the process can be adapted as follows: to remove, with probability  $p_{steal}$  the thread considers *stealing* tasks from a randomly chosen other queue, comparing the top element of a local queue with that on top of a globally-chosen random queue, and removing the higher-priority element. Otherwise, the thread directly removes from a local queue. As most insertions and deletions using the above scheme are local, this can result in a very unbalanced task distribution among queues, our analysis will adapt a stochastic scheduling model, by which threads are scheduled according to a scheduling distribution  $\vec{\pi} = (\pi_1, \pi_2, \dots, \pi_n)$ , where thread  $i$  is scheduled in each step with probability  $\pi_i$ , and we place upper and lower bounds on the maximum and minimum scheduling probabilities. Under these assumptions, we will be able to still provide rank guarantees.

A second performance issue with standard Multi-Queues is that, in practice, the overhead of inserting or removing a task can be large relative to the task execution time. The standard approach to address this issue is *task batching*, by which multiple tasks are inserted or removed at a single step. We show that the above random process can be resilient to task batching. Both the above approaches are well-known in the context of task scheduling *without priorities*, e.g. [9, 10], but have not been well-explored in the context of *priority* scheduling.

Our theoretical contribution is a generalization of the Multi-Queue analysis of [5] showing that, under assumptions, the above process, which we call *the stealing Multi-Queue (SMQ)*, induces a non-trivial trade-off between the stealing probability, the scheduling properties, and the average rank of elements removed. For instance, assuming a Multi-Queue formed of  $m = n$  queues (one queue per thread), with task batches of size  $O(B)$ , and a balanced thread scheduling distribution, we show that the expected rank removed at a step is  $O(Bm)$ , and  $O(Bm \log(Bm))$  with high probability. These bounds hold irrespective of the running time.

On the practical side, our work starts from an examination of the possible performance benefits of Multi-Queues relative to scheduling heuristics [20, 27], but also of their performance bottlenecks. The main benefit, which motivates our investigation, is the lower wasted work in real tasks, correlated to their rank guarantees. On the other hand, as mentioned, standard Multi-Queues have poor cache locality, relatively high per-task cost, and, so far, have had very limited specific implementation support. Our implementation addresses these shortcomings, via the following optimizations.

We begin by investigating the “optimal” data structure for implementing individual queues. While previous schedulers partitioned the tasks into sub-buffers, maintained either manually or semi-automatically by priority range, we adopt an efficient variant of a *sequential heap* for local task structure. To allow for efficient stealing, we affix a *stealing buffer* to each thread, into which the queue’s owner periodically places tasks, which can be either stolen by other threads or later processed by the queue owner. With this in place, we examine various mechanisms for allowing concurrent access to heaps and stealing buffers, and implement a task batching mechanism similar to the one described above.

This basic mechanism allows for several extensions, in particular a non-trivial NUMA-aware variant, which defines affinities and probabilities such that we seek to minimize out-of-socket accesses.

**Experimental Results.** We provide a general *SMQ* implementation on top of the Galois graph processing framework, which includes standard and NUMA-aware variants of the *SMQ*. Experiments show that our designs can surpass state-of-the-art schedulers such as OBIM and PMOD in terms of throughput and scalability when executing popular graph algorithms such as SSSP or A\*. Of note, much of the performance improvement comes from significantly less wasted work, which is linked to the improvement in rank guarantees provided by our scheduler.

**Related Work.** To our knowledge, the Multi-Queue-like data structure was given by the parallel branch-and-bound framework by Karp and Zhang [16], which distributed tasks randomly among queues, assigned to processors, and also remove tasks uniformly at random. We stress however that their proposal is in the context of classic task scheduling in the PRAM model, and that their design does not provide rank guarantees under asynchrony. Our construction starts from the Multi-Queue of Rihani, Sanders, and Dementiev [23], who introduced this design and provided a simple argument showing that the expected rank of the *first* removed element is  $O(m)$ . Follow-up work by Alistarh et al. [5] provided a more general and involved argument, showing that, for a sequential variant of the Multi-Queue, the expected rank of *any* removal is  $O(m)$ , by linking Multi-Queues with the classic  $(1 + \beta)$  random process of Peres, Talwar, and Wieder [21]. Follow-up work by the same authors [3] extended the analysis to a *concurrent* version of the Multi-Queue, under analytical assumptions.

Relative to this work, we adapt the standard Multi-Queue semantics so that they result in efficient implementations, in particular with respect to caching, add queue locality and task batching to the original design, and then adapt the analysis approach of [5] to prove rank bounds for the resulting algorithm.

There has been a tremendous amount of work on efficient scheduling heuristics for fine-grained task-based programs, especially in the context of graph processing [18, 13, 27, 20, 25, 11, 12]. A complete survey is beyond our scope, so we focus on the two works that are closest to ours. The first is [20], which details the design, implementation, and practical performance of the Galois system, focusing on the OBIM (Ordered By Integer Metric) scheduler. In brief, this scheduler assigns one *bag* (unordered set) per *task priority class*, which is empirically defined. Each bag is implemented as one or more FIFO queues, one per socket. Enqueues insert into the bag corresponding to the task priority, creating the bag if required, and each queue element is mapped to a batch (chunk) of tasks. Threads dequeue chunks from their socket’s queue; if that is empty, the thread steals from a remote queue. Tasks in a batch are performed one at a time. The list of bags is maintained in a global map, which is mirrored locally by each thread for cache efficiency.

Yesil et al. [27] start from the observation that the communication-avoiding pattern of OBIM can lead to significant wasted work, due to the relatively high number of priority inversions. Hence, they propose a heuristic which defines *priority groups*, which change dynamically at runtime. More precisely, PMOD tries to adapt the number of different priority bags in OBIM, by merging sets of similar priorities, so as to reduce the number of empty bags during runtime, thus trying to ensure that threads always have work to do. This is implemented via a dynamic merging mechanism. Conversely, the algorithm detects when there are *too few* priority bags, therefore splitting bags which are too full. These operations are controlled via carefully-designed heuristics.

Our design shares some features with these schedulers: in particular, we also recognize the importance of “localizing” the queues, and of batching for cache efficiency. However, it also differs in key ways, required to provide providing *rank guarantees* for the resulting scheduler. (None of these two previous heuristics have rank guarantees, and we do not believe that such guarantees could be shown without significant modifications.) For instance, in keeping with the Multi-Queue design, do not split tasks per priority “level,” and instead maintain a local heap structure for each queue.

Further, the stealing mechanism we use is different from both the standard Multi-Queue, and from the previous priority scheduling heuristics.

## 2 The Stealing Multi-Queue

### 2.1 The Classic Multi-Queue Design

The classic Multi-Queue uses  $m$  sequential queues, each protected by a lock, and distributes requests among them. Typically,  $m$  is taken to be the number of threads  $T$  multiplied by a constant factor  $C \geq 2$ , making it likely that individual operations will not interfere with each other when taking locks. An `insert(x)` comes, it chooses uniformly random queue and tries to lock it. When successful, it adds  $x$  into it and releases the lock. If the lock acquisition fails, the operation restarts. Similarly, `delete()` picks *two* different queues uniformly at random, and removes from the higher priority top element. Then, it tries to lock the chosen queue and retrieve the top task from it, releasing the lock at the end. If the lock acquisition fails, the operation restarts.

Listing 1 presents a pseudo-code of a simplified Multi-Queue version which may return `null` in `delete()` when the queue it removes an element from becomes empty. The version described here faithfully models the Galois implementation of Multi-Queues [20].

```

1 class MultiQueue<E> {
2   val queues = Queue<E>[C * T]
3
4   fun insert(task: E) = while(true) {
5     q := queues[random(0, queues.size)]
6     if !tryLock(q): continue
7     q.add(task)
8     unlock(q)
9     return
10  }
11
12  fun delete(): E? = while(true) {
13    i1, i2 := distinctRandom(0, queues.size)
14    q1 := queues[i1]; q2 := queues[i2]
15    if !tryLock(q1, q2): continue
16    q := q1.top() < q2.top() ? q1 : q2
17    task := q.extractTop()
18    unlock(q1, q2)
19    return element
20  }
21 }

```

Listing 1: The classic Multi-Queue implementation.

**Optimization 1: Task Batching.** A standard way to reduce the ratio between synchronization cost and task execution time in schedulers is *task batching*: `delete()` operations can retrieve multiple tasks from the same queue at once, storing them into a fixed-size thread-local buffer, and `insert(..)`s put the tasks into another thread-local buffer, flushing it to a random queue when the number of buffered tasks exceeds the buffer capacity. Most practical graph processing frameworks implement some variant of this optimization, e.g. [20]. Our analysis can bound its impact on the rank guarantees of the Multi-Queue.

The benefits of this approach are that (1) it reduces the number of lock acquisitions by a constant factor which is approximately the batch size and (2) it reduces the number of cache misses and contention compared to the version which accesses different queues on each `insert(..)` and `delete()`. However, it clearly impacts rank guarantees, since multiple elements are retrieved from the same queue so that other queues and further task insertions are “ignored” until the buffered tasks are processed. When this optimization is applied to insert operations, buffered tasks cannot be processed, so it also makes the implementation less fair.

**Optimization 2: Temporal Locality.** A similar, but different approach to reduce the cache coherence overhead is to use the same queue for a sequence of `delete()` or `insert()` operations, making it more likely that the corresponding data is already cached at the current core. Specifically, in this variant, the thread flips a biased coin before each new operation to decide whether to keep using the same queue as in the previous operation, or potentially pick another queue, according to the algorithm. (The coin is biased towards locality.) Our analysis approach can also provide rank bounds for this approach, although we will focus on analyzing the more performant *stealing* variant.

The difference between this method and task buffering is that updates to the queue (e.g. newly inserted tasks by another thread) would be visible to the current thread in this case. However, this approach is more costly, as it requires synchronization upon every operation, although obtaining the same lock multiple times, when uncontended, is relatively cheap.

We can examine the difference in terms of Dijkstra’s SSSP algorithm. In a step, the algorithm usually relaxes multiple edges, and thus adds several tasks to the queue. With temporal locality, the sequence of `insert(...)` invocations may insert part of all of these tasks into the same queue. Moreover, it is possible to insert several elements into the same queue with a single lock acquisition: with the lock acquired, the algorithm flips a coin after the insertion of each task to determine whether to keep inserting elements, or whether to potentially switch queues. With this optimization, the cost of lock acquisitions can be lower than in the classic Multi-Queue, but is still greater than with task buffering, where a fixed set of tasks is always inserted. The advantage, however, is in relatively better rank guarantees.

## 2.2 The Stealing Multi-Queue

The experimental data in Section 5 show that *task batching* and *temporal locality* optimizations can improve the performance of Multi-Queue scheduling for graph algorithms by up to  $3\times$  relative to the classic variant. However, accessing queues by different threads and lock acquisitions still have high performance cost. Therefore, we constructed a new variant which we call the *Stealing Multi-Queue (SMQ)*, which eschews locks, and improves cache locality beyond the optimizations discussed above, while maintaining rank guarantees under analytic assumptions.

Without task priorities, the classic way to implement schedulers, e.g. [10], is to use thread-local queues and allow *stealing*, so that threads can both add and retrieve tasks from their own queues, and steal tasks when the queues become empty.

The idea behind *SMQ* is similar — it also allows task stealing, but it uses thread-local *priority* queues. To guarantee fairness, *SMQ* steals tasks not only when it finds the thread-local queue empty, but also with a constant probability in each step. Specifically, in each step, with probability  $p_{steal}$ , the thread compares the priority of the top element in its local queue with that of a randomly chosen queue. In Section 3 we provide a rank analysis for this process under analytic assumptions.

Stealing lends itself to additional optimizations. For example, we employ the task batching optimization for task stealing: threads do not steal a single task at a time, but a whole batch. Intuitively, we aim to take advantage of the fact that in e.g. graph algorithms, tasks with similar priorities refer to nodes that are close to each other in the graph, and therefore it would be more efficient that they are processed together.

**High-Level Algorithm.** Listing 2 presents a high-level pseudocode for the *SMQ* algorithm. Thread-local queues are stored in the `queues` array, where `queues[t]` is associated with thread `t` (line 3). Additionally, each threads owns a `stolenTasks` buffer of capacity  $SIZE_{steal} - 1$  that is thread-local and stores the tasks stolen from another queue (line 4).

```

1 class StealingMultiQueue<E> {
2   // queues[t] is associated with thread 't'
3   val queues: Queue<E>[threads]
4   threadlocal val stolenTasks = Buffer<E>(SIZEsteal-1)
5
6   fun insert(task: E) =
7     queues[curThread()].addLocal(task)
8
9   fun delete(): E? {
10    // Do we have previously stolen tasks?
11    if stolenTasks.isNotEmpty():
12      return stolenTasks.removeFirst()
13    // Should we steal?
14    with psteal probability {
15      task := trySteal()
16      if task != null: return task
17    }
18    // Try to retrieve the top task
19    // from the thread-local queue
20    task := queues[curThread()].extractTopLocal()
21    if (task != null) return task
22    // The local queue is empty, try to steal
23    return trySteal()

```

```

24 }
25
26 fun trySteal(): T? {
27     // Choose a random queue and check whether
28     // its top task has higher priority
29     t := curThread()
30     qId := random(0, queues.size)
31     if queues[qId].top() < queues[t].top():
32         // Try to steal a better task!
33         stolen := queues[qId].steal(STEAL_SIZE)
34         if stolen.isEmpty(): return null // failed
35         // Return the first task and add the others
36         // to the thread-local buffer of stolen ones
37         stolenElements.add(stolen[1:])
38         return stolen[0]
39     }
40 }

```

Listing 2: High-level Stealing Multi-Queue (*SMQ*) algorithm. It assumes that Queue is provided with an additional `steal(k)` function that retrieves top  $k$  tasks (or less, if the queue size is lower). The implementation of this `steal(k)` function is discussed in Section 4.

The `insert(...)` operation is straightforward—it simply adds the specified task into the thread-local queue (lines 6–7).

The `delete()` operation first processes any buffered stolen tasks (lines 11–12). In case the buffer is empty, it steals tasks from another queue with probability  $p_{steal}$  and returns the stolen task of highest priority if this succeeds (lines 14–17). If the algorithm did not steal or stealing has failed (the `trySteal()` invocation at line 15 returned `null`), `delete()` retrieves the top task from the thread-local queue and returns it (lines 20–21). However, the thread-local queue can be empty. In this case, the algorithm tries to steal tasks from another queue (line 23). As before, this implementation may return `null` if not task is found.

The stealing logic is implemented in the `trySteal()` function (lines 26–39) which attempts to steal  $SIZE_{steal}$  tasks from a random queue if the top element from the thread-local queue has lower priority, and returns the top task as a result, storing all the resulting tasks into the thread-local `stolenTasks` buffer.

### 3 Analysis of the Stealing Multi-Queue

We consider the following simplified variant of the *SMQ* algorithm in the analysis.

```

1 class StealingMultiQueue<E> {
2     // queues[t] is associated with thread 't'
3     val queues: Queue<E>[threads]
4
5     fun insert(task: E) =
6         queues[curThread()].add(task)
7
8     fun delete(): List<E> {
9         with  $p_{steal}$  probability { // should we steal?
10             return trySteal()
11         }
12         // Never empty in the theoretical model
13         return queue[curThread()].extractTopB()
14     }
15
16     fun trySteal(): List<E> {
17         // Choose a random queue and check whether its
18         // top task has higher priority,  $\Rightarrow$  lower rank
19         t := curThread()
20         qId := random(0, queues.size)
21         if (queues[qId].top() < queues[t].top()):
22             return queues[qId].extractTopB()
23         }
24         return queues[t].extractTopB()

```

Listing 3: The stealing Multi-Queue algorithm assumed in the analysis. It assumes that each queue is equipped with `extractTopB()` method which retrieves and returns its top  $B$  elements.

**Analytical Model.** We now provide rank bounds for the above algorithm, in a simplified analytical model. As in [5], we assume that all element insertions occur initially and in increasing rank order. Additionally, we assume that tasks are inserted at random among the queues, and that `extractTopB()` operations occur atomically, so that we can analyze a sequential “linearized” version of the process.

Importantly, we assume a stochastic scheduling model similar to that of [8], where we are given a thread scheduling distribution  $\vec{\pi} = (\pi_1, \pi_2, \dots, \pi_n)$  for the  $n$  threads, where  $\pi_i$  is the probability with which thread  $\pi_i$  is scheduled to perform operation in a step. Since we assume that all insertions occur initially, threads will not observe empty queues, so our main objective will be to characterize the expected rank bound of the elements removed, relative to all elements present in the queues. The `extractTopB()` operation retrieves  $B$  elements, for constant  $B$ ; however, for convenience, we will first describe the case  $B = 1$  first.

Let  $p_{steal}$  be the stealing probability, that is: once a thread is scheduled to perform a delete operation, with probability  $p_{steal}$  stealing occurs: it picks a second queue uniformly at random among all queues, and “steals” its top element if the rank of its top element is smaller than the rank of the element on top its local queue. If no stealing occurs, the thread simply deletes the top element of its local queue and returns it. Finally, we assume that there exists a constant  $\gamma$  such that, for each thread  $i$ ,  $1 - \gamma \leq \frac{1}{\pi_i n} \leq 1 + \gamma$ , for  $\gamma \leq 1/2$ . Intuitively, the parameter  $\gamma$  bounds how unfair the thread scheduler may be. For instance, the value  $\gamma = 0$  means that the thread scheduler is completely uniform.

**The Main Theorem.** Given the above definitions, our main claim is the following.

**Theorem 1.** *Assume the thread scheduling probability distribution  $\vec{\pi} = (\pi_1, \pi_2, \dots, \pi_n)$  with constant  $\gamma \geq 0$  such that, for each thread  $i$ ,  $1 - \gamma \leq \frac{1}{\pi_i n} \leq 1 + \gamma$ , for  $\gamma \leq 1/2$ ,<sup>1</sup> and let  $p_{steal}$  be the stealing probability. If  $\gamma \left( \frac{1}{p_{steal}} - 1 \right) \leq \frac{1}{2n}$ , then the SMQ process which removes  $B$  elements during the delete operation satisfies that, for any time step  $t$ , the expected maximum rank of elements on top of queues is  $O\left(\frac{nB(1+\gamma)}{p_{steal}} \left(\log n + \log \frac{(1+\gamma)}{p_{steal}}\right)\right)$  and the expected average rank (maximum and average ranks are computed over  $Bn$  elements:  $B$  elements on top of  $n$  queues) is at most  $O\left(\frac{nB(1+\gamma)}{p_{steal}} \log \frac{(1+\gamma)}{p_{steal}}\right)$ .*

**Discussion.** The above claim is somewhat abstract, so let us build intuition via some examples. Consider first the case of a *uniform* scheduling distribution,  $\gamma = 0$ , and no task batching,  $B = 1$ . Then, if the stealing probability is *constant*, the expected average rank is  $O(n)$ , and the expected maximum rank is  $O(n \log n)$ . If we wish to steal less often, i.e.  $p_{steal} = O(1/n)$ , then the expected rank cost becomes  $O(n^2 \log n)$  in both cases.

More generally, the claim also shows that, if  $p_{steal}$  is large enough, we consistently obtain good rank bounds. Given parameter  $0 \leq q \leq 1$ , if  $p_{steal} \geq 1 - \frac{1}{n^q + 1} \geq \frac{1}{2}$  and  $\gamma \leq \frac{n^q}{2n}$ , then  $\gamma \left( \frac{1}{p_{steal}} - 1 \right) \leq \frac{1}{2n}$  and  $\frac{1+\gamma}{p_{steal}} = O(1)$ . Hence, the expected average rank bound becomes  $O(n)$ , irrespective of the scheduling imbalance. We can also use smaller  $p_{steal}$  at the expense of larger rank bounds and smaller tolerated  $\gamma$ . If  $\gamma \leq \frac{p_{steal}}{2n} \leq \frac{1}{2n}$ , threads are scheduled almost uniformly. Here, the expected average rank bound becomes  $O\left(\frac{n \log \frac{1}{p_{steal}}}{p_{steal}}\right)$ .

**Analysis Overview.** Our analysis generalizes the argument of [5] to the more intricate variant of the Multi-Queue process which we consider. Due to space constraints, we only sketch the argument here, and provide the full proof as supplementary material.

The first step is similar to [5]: we describe a coupling which equates the discrete SMQ process described above to a *continuous* balls-into-bins process. We replace the  $n$  queues with  $n$  bins, each of which initially contains a single ball of label 0. We start by inserting infinitely many balls into the bins, so that for each bin  $i$  the difference between the labels of two consecutive balls is an exponential random variable with mean  $\pi_i$ . Then, we perform  $T$  insertions in the modified SMQ process (recall that elements are inserted in the increasing rank order) as follows : for each element with rank  $t \leq T$ , we insert it into  $i$  if the label (ball) with rank  $t$  is inserted in the bin  $i$ . Finally, we remove the labels which have rank larger than  $T$  from the bins. Note that after the insertion phase the rank distributions of labels (balls)

<sup>1</sup>If the scheduling distribution is *uniform*, then  $\gamma = 0$ .

in bins and elements in queues are equal. The first technical step is a Lemma proving that elements are inserted into the queues in the same way as the original *SMQ* process.

Assuming that the number of initial insertions  $T$  is large, we extend the coupling to the second *removal phase*. In each removal step, we first pick a “local” bin  $i$  with probability  $\pi_i$ , and then flip a coin to decide whether to steal or not. With probability  $p_{steal}$ , we decide to steal. If so, we pick a second bin uniformly at random from among all  $n$ , and examine the two balls of lowest label (highest priority) on “top” of the two bins. Following the priority process, we remove the ball of lowest label among the two, uncovering the next ball in the bin. If the coin flip dictated that we *not steal*, then we directly remove the ball on top of bin  $i$ . (Notice that, in both cases, the label increment for a chosen bin is exponentially-distributed). If batch size  $B > 1$ , we remove  $B$  labels from the bin. In the *SMQ* process we make the same random choices as in the balls and bins process and follow the same removal procedure, and since the rank distributions are equal after the insertion phase, it is easy to see that they will stay equal after every removal. That is, if the ball is removed from the bin  $k$  then the element is removed from the queue  $k$ .

The *rank cost* at a step is the *rank* of the label of the removed ball among all labels still present in all the bins. Because of the coupling, this will imply that the rank cost in the *SMQ* process is also bounded in expectation.

Let  $\ell_i(t)$  be the label on top of bin  $i$  at time step  $t$ . Let  $x_i(t) = \ell_i(t)/n$  be the normalized value of this label, and let  $\mu(t) = \frac{1}{n} \sum_{i=1}^n x_i(t)$  be the average normalized label at time step  $t$ . As in [21, 5], we will be analyzing the potential function

$$\Gamma(t) = \sum_{i=1}^n e^{-\alpha(x_i(t) - \mu(t))} + \sum_{i=1}^n e^{\alpha(x_i(t) - \mu(t))}, \quad (1)$$

for a suitable constant  $\alpha > 0$ , which we will define later.

A key ingredient of this analysis is the  $(1 + \beta)$ -choice random process [21], which is similar to the continuous process we defined above, with the difference that in order to delete an element, with probability  $(1 - \beta)$  we choose a single bin uniformly at random and delete from it, and with probability  $\beta$  we choose two bins uniformly at random and delete from the one which has a ball of lower label on top. (Thus, in expectation, we perform  $(1 + \beta)$  choices at a step.)

**Bounding the Potential.** Fix an arbitrary time step  $t$ , and the labels  $x_1(t), x_2(t), \dots, x_n(t)$  on top of the bins. Let  $\Gamma_c(t + 1)$  be the potential at time step  $t + 1$  if the label is deleted by our continuous process, and let  $\Gamma_\beta(t + 1)$  be the potential at time step  $t + 1$  if the label is deleted by  $1 + \beta$  process. Our strategy is to show that there exists  $\beta$  such that

$$\mathbb{E}[\Gamma_c(t + 1) | (x_i(t))_{i=1, n}] \leq \mathbb{E}[\Gamma_\beta(t + 1) | (x_i(t))_{i=1, n}].$$

At this point, our argument diverges from that of [5]. First, we define  $S_c(i)$  to be the probability that we delete a label from one of the first  $i$  bins in our process continuous process, and let  $S_\beta(i)$  be the same probability for the  $(1 + \beta)$  process. Our first technical step is to show that, under reasonably chosen parameter values  $\gamma \left( \frac{1}{p_{steal}} - 1 \right) \leq \frac{1}{2n}$  and  $\beta = \frac{p_{steal}}{2(1+\gamma)}$ , we have that for any  $1 \leq i \leq n$ ,  $S_c(i) \geq S_\beta(i)$ .

Based on this, our key technical step will be to use coupling in order to show that the potential dominance claimed above holds for suitable chosen parameter values. Subsequently, we can use [5, Lemma 3] to show that for any step  $t + 1$ :

$$\begin{aligned} \mathbb{E}[\Gamma_c(t + 1) | (x_i(t))_{i=1, n}] &\leq \mathbb{E}[\Gamma_\beta(t + 1) | (x_i(t))_{i=1, n}] \\ &\leq \left( 1 - \Omega \left( \frac{\beta^2}{n} \right) \right) \Gamma(t) + \text{poly} \left( \frac{1}{\beta} \right). \end{aligned}$$

The rest of the proof leverages the implied potential bound to obtain *rank* bounds on the elements removed. First, we use the above supermartingale type bound to prove that for any time step  $t \geq 0$ :  $\mathbb{E}[\Gamma_c(t)] \leq O \left( \text{poly} \left( \frac{1}{\beta} \right) n \right)$ . Finally, we can use Theorems 4 and 5 in the full version of [5] to show that for the continuous process and time  $t \geq 0$ , the expected maximum rank of labels on top of bins at time step  $t$  is  $O \left( \frac{n}{\beta} (\log n + \log \frac{1}{\beta}) \right)$  and expected average rank is  $O \left( \frac{n}{\beta} \log \frac{1}{\beta} \right)$ . Plugging in  $\beta = \frac{p_{steal}}{2(1+\gamma)}$  in the above rank provides the proof of the theorem for  $B = 1$ . The proof for  $B > 1$  is slightly more involved, and is described in the supplementary material.

## 4 SMQ Implementation Details

In Section 2.2 we presented the general SMQ design, skipping the details of the stealing implementation. Here, we address this gap. In our investigation, we have first used *concurrent skip-lists* as thread-local queues as well as *sequential d-ary heaps* augmented with special *stealing buffers*. We have found the latter approach to perform consistently better, and therefore we focus on it here.

**SMQ via d-ary Heaps with Stealing Buffers.** Listing 4 presents the pseudo-code for this design. We use sequential *d*-ary heaps (typically, with parameter  $d = 4$ ) as thread-local queues for storing tasks and separate the synchronization and stealing from the sequential heap implementation.

```
1 class HeapWithStealingBufferQueue<E> {
2   val q = Heap<E>() // local d-ary heap
3   // Other threads can steal from this buffer
4   val stealingBuffer = Buffer<E>(SIZEsteal)
5   // 64-bit register with the current buffer epoch
6   // and the "tasks are stolen" flag
7   val (epoch, stolen): (Int, Bool) = (0, true)
8
9   fun addLocal(task: E) {
10    q.add(task) // add to the local queue
11    if stolen: fillBuffer()
12  }
13
14  fun extractTopLocal(): E? {
15    if stolen: fillBuffer()
16    return q.extractTop()
17  }
18
19  fun top(): E? = while(true) {
20    // Read the current epoch and the flag
21    (curEpoch, curStolen) := (epoch, stolen)
22    if stolen: return null // can we steal?
23    top := stealingBuffer.first() // read the top
24    if curEpoch != epoch: continue // restart
25    return top // return the top buffer element
26  }
27
28  fun steal(size: Int): List<E> = while(true) {
29    // Read the current epoch and the flag
30    (curEpoch, curStolen) := (epoch, stolen)
31    if stolen: return emptyList() // can't steal
32    // Read the tasks non-atomically
33    tasks := stealingBuffer.read()
34    atomic { // atomically update (epoch, stolen)
35      if epoch != curEpoch || stolen: continue
36      stolen = true // the tasks have been stolen!
37    }
38    return tasks
39  }
40
41  fun fillBuffer() { // stolen == true
42    stealingBuffer.clear()
43    // Fill the buffer, keep the flag 'true'
44    repeat(STEAL_SIZE) {
45      task := q.extractTop()
46      if task == null: break
47      stealingBuffer.add(task)
48    }
49    // Increment the epoch and re-set the flag
50    (epoch, stolen) = (epoch + 1, false)
51  }
52 }
```

Listing 4: Stealing Multi-Queue implementation via *d*-ary heaps with stealing buffers.

To steal efficiently, we maintain the metadata such as a buffer epoch and a “tasks are stolen” flag in a single 64-bit integer field (line 7). When the tasks in the buffer are stolen, the flag should be set to `true`. Simultaneously, when the buffer is filled with new tasks, the epoch increases.

Thus, the stealing procedure presenting in the `steal(...)` function (lines 28–39) is organized as follows. First, it atomically reads the current epoch and the flag (line 30). If the tasks have been already stolen, the function fails and returns an empty list (line 31). Next, it reads the buffer in a non-atomic way in the hope that the tasks are still in the buffer (line 33). Then, the algorithm atomically checks that the epoch has not been updated, and changes the flag from `false` to `true` (line 34–37). On success, it returns the already read tasks; otherwise, it restarts the procedure from the beginning.

In the `add(...)` operation, the task is added to the thread-local heap (line 10) followed by a `fillBuffer()` invocation if the tasks from the buffer are stolen (line 11). The `extractTop()` operation also delegates the work to the sequential heap, filling the buffer with new tasks if needed (lines 14–17).

The `top()` operation uses the same approach as stealing: it reads the current epoch and flag (line 21), checks that the tasks are not stolen (line 22), reads the top task from the buffer (line 23), and checks that the epoch has not been changed (line 24). When the epoch is the same, the operation returns the already read task; otherwise, it restarts.

To refill the buffer (lines 51–41), the algorithm clears it first (line 42) and puts  $\text{SIZE}_{steal}$  top tasks from the local sequential heap (lines 44–48). During this procedure, the “tasks are stolen” flag is `true`, so other threads do not interfere. In the end, the algorithm applies refilling by atomically increasing the epoch and resetting the flag (line 50).

**NUMA-Awareness.** The classic Multi-Queue design does not consider Non-Uniform Memory Access (NUMA) effects, and threads are likely to access queues located on another socket, which can affect performance. We present a simple strategy which significantly reduces overheads, and fits the analysis of Section 3.

Assume  $N$  NUMA nodes with  $T_i$  threads each (here,  $i$  is the node index). In this configuration, we create  $T_i \times C$  queues for each node. This way, the total number of queues stays the same as before. The straightforward idea is for threads to preferentially access queues from the same node, so all the synchronization conflicts are likely to be resolved on the last-level cache (usually, L3) of the current NUMA node. For this, we use a weighted probability distribution to be used when choosing a new queue to sample as part of either the Multi-Queue or the Stealing Multi-Queue. For a given thread, all  $T_i \times C$  queues associated with its own node will have weight 1 while all other queues, associated with other nodes, will have weight  $\frac{1}{K}$ , where  $K > 1$  is a constant. Intuitively, by adjusting the multiplier  $K$ , we balance fairness with the number of “out-of-node” accesses.

Specifically, given a thread from node  $i$ , the total weight of all queues is  $W_i = \sum_{j \neq i} \frac{T_j \cdot C}{K} + T_i \times C$ . The probability of choosing a queue from the same node by a thread from the node  $i$  is  $T_i C / W_i$ . Thus the expected number of “internal” queue choices by all threads in node  $i$  is  $E_i = T_i^2 \cdot C / W_i$ , and the expected ratio of “internal” queue choices by *all* threads in *all* nodes is  $E = \sum E_i$ . This metric represents the “NUMA-friendliness” of the algorithm. Typically, since we use all cores, the number of threads on each NUMA node is equal to  $T/N$ , where  $T = \sum_i T_i$ . When  $K > N$ , which is reasonable in practice, we simplify to the expression of  $E$  to

$$E_{int} \approx T \times \left(1 - \frac{1}{K}\right).$$

In our implementation, we aim to keep this ratio constant as we increase the thread count, and therefore we make  $K$  depend linearly depend on the number of threads  $T$ .

## 5 Evaluation

We now examine how the various optimizations we discussed impact the performance of the Multi-Queue, and compare relative to high-performance scheduling heuristics, such as OBIM and its adaptive PMOD version, Spray-List [6], and the random-enqueue local-dequeue (RELD) algorithm from [14]. We examine throughput versus a single-threaded baseline (and hence also scalability) but also the total amount of additional work incurred due to scheduling. We build on Galois [20], a popular graph processing framework, which is still maintained and provides efficient implementations of the above two schedulers. Our implementation will be made open-source upon publication.

**Hardware.** Experiments were performed on an AMD machine featuring two EPYC 7702 64-Core processors with hyper-threading for 256 total hardware threads, and an Intel machine with features four Intel Xeon Gold 5218 processors with 16 cores each for 128 hardware threads in total.

Graph	V	E	Description
USA	24M	58M	Full USA roads
WEST	6M	15M	Roads of the western part of the USA
TWITTER	41M	1468M	Directed graph describing follower relation in Twitter
WEB	50M	1930M	Directed web crawl of .sk domain

Table 1: Input graphs for the algorithms used to benchmark different schedulers.

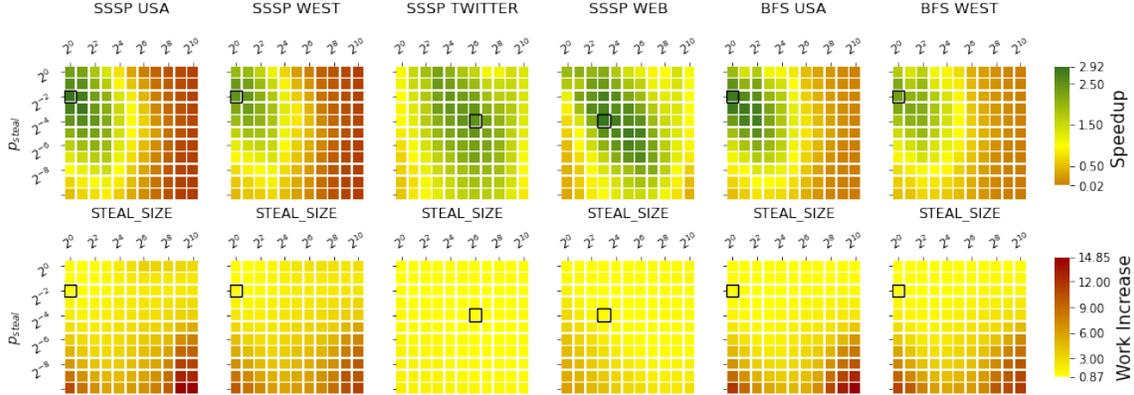


Figure 1: Ablation of stealing probability  $p_{steal}$  and steal buffer size, for  $SMQ$  implemented using  $d$ -ary heaps, relative to wasted work, for a subset of benchmarks. Experiments are executed on the AMD machine on 256 threads. The baseline is the Multi-Queue on 256 threads with  $C = 4$ . The fastest configuration for each benchmark is highlighted with a black border. Best viewed in color.

**Benchmarks.** We use the real-world road and social graphs listed in Table 1 for experiments. The first two graphs represent the full USA road network and its West part. The second two graphs represent follower relations in the Twitter social network and a web crawl of .sk domain; edge weights are uniform random in the range  $[0, 255]$ .

We use various graph algorithms to test schedulers under different workloads: **Single-Source Shortest Paths (SSSP):** The Galois implementation of SSSP based on delta-stepping. We evaluate it on the USA and WEST road graphs, and on TWITTER and WEB social network graphs. **Breadth-First Search (BFS):** The classic traversal algorithm a graph, where the weight of each edge is 1, evaluated on USA and WEST road graphs, and on TWITTER and WEB social network graphs. **A\*:** This algorithm calculates the distance between two vertices, guided by the expected distance to the destination vertex from the currently visiting one. As a heuristic, the equi-rectangular approximation is used. We evaluate it on the USA and WEST road graphs. **Minimum Spanning Tree (MST):** We use Boruvka’s algorithm to find a spanning tree over all vertices with minimum total edge weight, with task priority equal to the degree of the associated vertex. We evaluate it on the USA and USA-WEST graphs.

**Metrics.** For all runs, we record end-to-end times for the given tasks, as well as total number of tasks executed, to measure wasted work relative to the baseline. We use 10 repetitions, and show the average. (Standard deviation is low, so we omit confidence intervals for readability.)

**Methods and Tuning.** We use the PMOD and OBIM baselines provided as part of Galois [20]. We follow PMOD [27] for parameter and benchmark setups. Specifically, we start from their optimized choices for the  $\Delta$  and  $CHUNK\_SIZE$  parameters, for both OBIM and adaptive PMOD schedulers, and perform additional tuning to maximize throughput on our setup. Full experimental data is presented in Appendix Section B. We also examine variants of the classic Multi-Queue, including Random Enqueue Local Dequeue (RELD), with and without the suggested buffering and locality optimizations. (In brief, we found that these optimizations, without stealing, improved the throughput of the baseline implementation by up to  $3.4\times$ .) Further, we implemented the  $SMQ$  proposal described in Section 2 both with local skip-lists, and using local heaps. We also implemented the buffering optimization.

**Impact of Optimizations and Parameters.** Our first set of experiments aims to examine the impact of the *task batching* and *temporal locality* optimizations on the performance of Multi-Queue-based proposals. We begin with the classic Multi-Queue and its variants. We found that most variants “scale,” in the sense that their maximal throughput

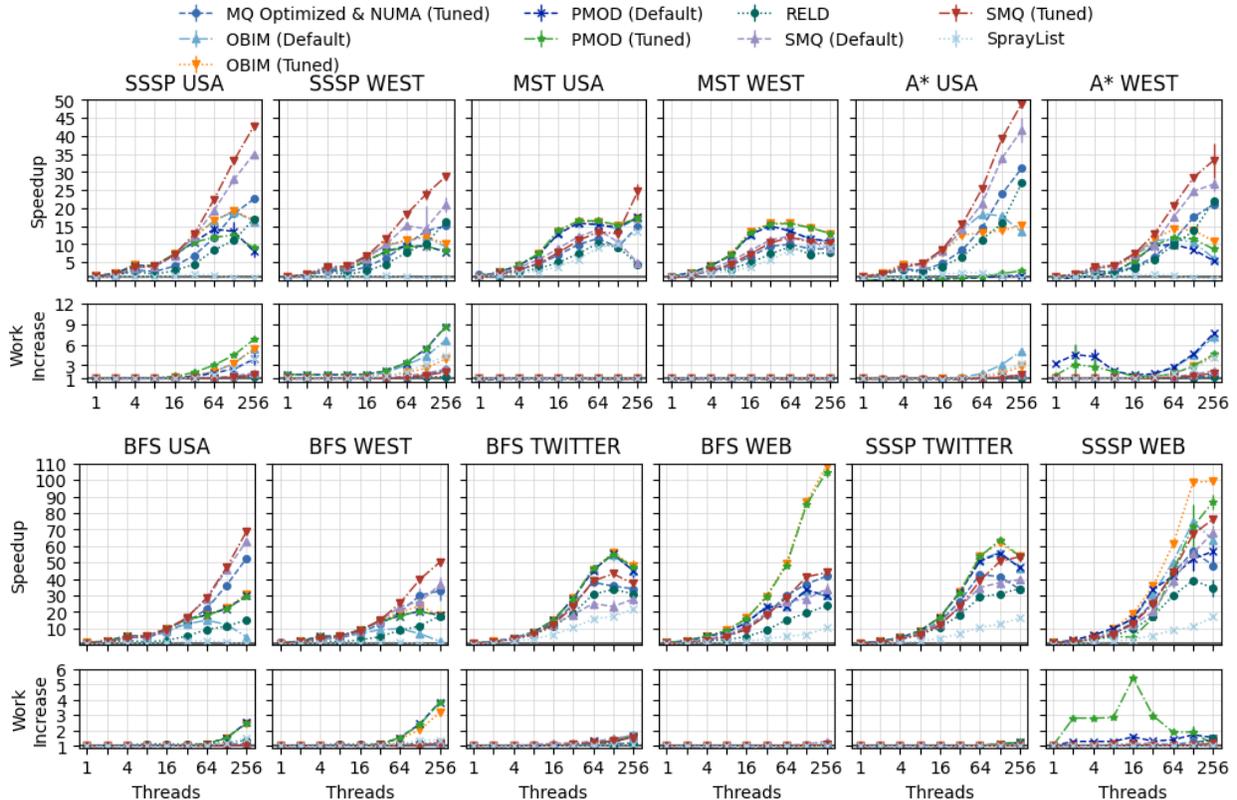


Figure 2: Comparison between the best and the default variants of SMQ, optimized classic MQ, tuned OBIM, PMOD, and other advanced schedulers on the AMD platform. Speedups are versus the baseline Multi-Queue, running on a single thread. See text for full details.

is achieved at the highest number of threads. (Lower performance after 128 threads is explained by the fact that hyper-threads share their cache, thus technically halving its size.) Generally, our optimizations significantly improve the performance of the Multi-Queue by up to  $3.4\times$ . Appendix C presents a set of experiments on both AMD and Intel platforms, evaluating in terms of wall-clock speedup, and total number of tasks performed, relative to the sequential baseline.

Next, we examine the impact on the various parameter values, in particular *local queue implementation*, *stealing buffer size* and *stealing probability*, on the performance of the *Stealing Multi-Queue (SMQ)*. Figure 1 presents results for the heap-based SMQ, in terms of heatmaps examining *speedup* (top row) and *wasted work* (work increase) for the various parameter values. (Appendix D contains a full set of results on all graphs and on both AMD and Intel platforms, also examining the skip-list variant.) We observe that the speedups are fairly stable across parameter values, except at very high batch sizes, and that loss of performance correlates well with increase in number of tasks (work) performed.

**Comparison with PMOD and OBIM.** Figure 2 presents the results of the speedup and total work comparison between the SMQ (both using heaps and skip-lists), the best-performing variant of the optimized Multi-Queue with NUMA-aware sampling, PMOD, OBIM, RELD, and Spray-List schedulers, executed on the AMD machine. (Please see the Appendix F for magnified versions of this graph on both AMD and Intel platforms.) We present two variants of the SMQ with respect to the tuning process. SMQ (Tuned) adopts parameter values derived via task-specific tuning (see Figure 1), while SMQ (Default) picks a set of reasonable default parameters ( $STEAL\_SIZE = 4$ ,  $p_{steal} = 1/8$ ,  $K = 8$ ) across all benchmarks.

We note the following. First, an examination of the throughput graphs (top) shows that the SMQ provides the highest throughput at 256(128 for Intel) threads in 10 out of the 12 experiments, and is virtually tied in the 6th experiment (MST). The only experiments where it has lower overall performance relative to OBIM and PMOD is the BFS experiment on social network graphs. The simple explanation is that, here, throughput is more important than task ordering: the task priorities are “flat” due to the graph having high expansion, and there is essentially no difference in terms

of the total number of tasks performed by the different schedulers. On all other tasks, we find that the SMQ provides similar or higher performance relative to scheduling heuristics, of up to  $1.84\times$ . Focusing on the SSSP and A\* experiments, the main reason is the lower number of total tasks performed, as well as a relatively low per-operation cost due to batching and the stealing buffer implementation. Of note, the number of tasks performed appears to be near-constant with respect to the number of threads/queues. We further emphasize that this finding is valid even in the absence of task-specific tuning.

Next, we notice that, perhaps surprisingly, even the standard Multi-Queue can provide competitive results on a range of benchmarks, as long as it uses the optimized and NUMA-aware variants. However, its throughput does not surpass that of the SMQ, motivating the stealing optimization. Finally, we note that both OBIM and PMOD provide very competitive results, and good scalability in all benchmarks. We encourage the reader to examine the Appendix for additional experiments and ablation studies.

## 6 Discussion

We presented an in-depth investigation of scalable priority scheduling for graph algorithms, focusing on a new Multi-Queue variant which we show to be competitive with state-of-the-art scheduling heuristics, while still providing theoretical guarantees. In future work, we plan to examine comparisons with alternative parallelization approaches [12], and other applications, such as iterative machine learning algorithms e.g. [2].

## References

- [1] Maleen Abeydeera, Suvinay Subramanian, Mark C Jeffrey, Joel Emer, and Daniel Sanchez. Sam: Optimizing multithreaded cores for speculative parallelism. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 64–78. IEEE, 2017.
- [2] Vitalii Aksenov, Dan Alistarh, and Janne H Korhonen. Scalable belief propagation via relaxed scheduling. *Advances in Neural Information Processing Systems*, 33, 2020.
- [3] Dan Alistarh, Trevor Brown, Justin Kopinsky, Jerry Z Li, and Giorgi Nadiradze. Distributionally linearizable data structures. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, pages 133–142, 2018.
- [4] Dan Alistarh, Trevor Brown, Justin Kopinsky, and Giorgi Nadiradze. Relaxed schedulers can efficiently parallelize iterative algorithms. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, pages 377–386, 2018.
- [5] Dan Alistarh, Justin Kopinsky, Jerry Li, and Giorgi Nadiradze. The power of choice in priority scheduling. *arXiv preprint arXiv:1706.04178*, 2017. In Proceedings of PODC 2017.
- [6] Dan Alistarh, Justin Kopinsky, Jerry Li, and Nir Shavit. The spraylist: A scalable relaxed priority queue. In *20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2015, San Francisco, CA, USA, 2015*. ACM.
- [7] Dan Alistarh, Giorgi Nadiradze, and Nikita Koval. Efficiency guarantees for parallel incremental algorithms under relaxed schedulers. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, pages 145–154, 2019.
- [8] Dan Alistarh, Thomas Sauerwald, and Milan Vojnović. Lock-free algorithms under stochastic schedulers. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, pages 251–260, 2015.
- [9] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *ACM SigPlan Notices*, 30(8):207–216, 1995.
- [10] Robert D Blumofe and Charles E Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.
- [11] Laxman Dhulipala, Guy Blelloch, and Julian Shun. Julienne: A framework for parallel graph algorithms using work-efficient bucketing. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 293–304, 2017.
- [12] Laxman Dhulipala, Guy E Blelloch, and Julian Shun. Theoretically efficient parallel graph algorithms can be fast and scalable. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, pages 393–404, 2018.

- [13] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, volume 12, page 2, 2012.
- [14] Mark C Jeffrey, Suvinay Subramanian, Cong Yan, Joel Emer, and Daniel Sanchez. A scalable architecture for ordered parallelism. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 228–241, 2015.
- [15] Mark C Jeffrey, Suvinay Subramanian, Cong Yan, Joel Emer, and Daniel Sanchez. Unlocking ordered parallelism with the swarm architecture. *IEEE Micro*, 36(3):105–117, 2016.
- [16] R. M. Karp and Y. Zhang. Parallel algorithms for backtrack search and branch-and-bound. *Journal of the ACM*, 40(3):765–789, 1993.
- [17] A Lenharth, D Nguyen, and K Pingali. Concurrent priority queues are not good priority schedulers. In *Proceedings of the 21th International European Conference on Parallel and Distributed Computing*, pages 209–221, 2015.
- [18] Yucheng Low, Joseph E Gonzalez, Aapo Kyrola, Danny Bickson, Carlos E Guestrin, and Joseph Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1408.2041*, 2014.
- [19] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, SOSP '13*, pages 456–471, New York, NY, USA, 2013. ACM.
- [20] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, SOSP '13*, pages 456–471, New York, NY, USA, 2013. ACM.
- [21] Yuval Peres, Kunal Talwar, and Udi Wieder. Graphical balanced allocations and the 1 + beta-choice process. *Random Struct. Algorithms*, 47(4):760–775, December 2015.
- [22] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, et al. The tao of parallelism in algorithms. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 12–25, 2011.
- [23] Hamza Rihani, Peter Sanders, and Roman Dementiev. Brief announcement: Multiqueues: Simple relaxed concurrent priority queues. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '15*, pages 80–82, New York, NY, USA, 2015. ACM.
- [24] Konstantinos Sagonas and Kjell Winblad. The contention avoiding concurrent priority queue. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 314–330. Springer, 2016.
- [25] Julian Shun and Guy E Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 135–146, 2013.
- [26] Martin Wimmer, Jakob Gruber, Jesper Larsson Träff, and Philippas Tsigas. The lock-free k-lsm relaxed priority queue. In *ACM SIGPLAN Notices*, volume 50, pages 277–278. ACM, 2015.
- [27] Serif Yesil, Azin Heidarshenas, Adam Morrison, and Josep Torrellas. Understanding priority-based scheduling of graph algorithms on a shared-memory platform. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2019.

## A Full Analysis

**General Approach.** Our analysis will generalize the argument of [5] to the more intricate variant of the Multi-Queue process which we consider. The first step in both analyses is similar: we describe coupling which equates the discrete  $SMQ$  process described above to a *continuous* balls into bins process.

We start with the insertion phase. Imagine we replace the  $n$  queues with  $n$  bins, each of which initially contains a single ball of label 0. We start by inserting infinitely many balls into bins, so that for each bin  $i$  the difference between the labels of two consecutive balls is an exponential random variable with mean  $\pi_i$ . Then, we perform  $T$  insertions in the modified  $SMQ$  process (recall that elements are inserted in the increasing rank order) as follows : for each element with rank  $t \leq T$ , we insert it into the queue  $i$  if the label(ball) with rank  $t$  is inserted in the bin  $i$ . Finally, we remove the labels which have rank larger than  $T$  from the bins. Note that after the insertion phase the rank distributions of labels (balls) in bins and elements in queues are equal. In order to show that the coupling is valid we need the following Lemma, which proves that elements are inserted into the queues in the same way as the original  $SMQ$  process.

**Lemma 1** (Coupling, Theorem 2 in [5]). *Let  $I_{i \leftarrow j}$  be the event that the label with rank  $j$  is located in bin  $i$  and let  $Pr[I_{i \leftarrow j}]$  be its probability. We have that  $I_{i \leftarrow j}$  is independent from  $I_{i' \leftarrow j'}$  for all  $j \neq j'$  and  $Pr[I_{i \leftarrow j}] = \pi_i$ .*

Assuming that the number of initial insertions- $T$  is large, we describe how coupling works in a second *removal phase*. In each removal step, we first pick a “local” bin  $i$  with probability  $\pi_i$ , and then flip an additional coin to decide whether to steal or not. With probability  $p_{steal}$ , we decide to steal. If so, we pick a second bin uniformly at random from among all  $n$ , and examine the two balls of lowest label (highest priority) on “top” of the two bins. Following the priority process, we remove the ball of lowest label among the two, uncovering the next ball in the bin. If the coin flip dictated that we *not steal*, then we directly remove the ball on top of bin  $i$ . (Notice that, in both cases, the label increment for a chosen bin is exponentially-distributed). When batch size  $B > 1$ , we remove  $B$  labels from the bin, but for the simplicity we are going to deal with the  $B = 1$  case first. In the  $SMQ$  process we make the same random choices as in the balls and bins process and follow the same removal procedure, and since the rank distributions are equal after the insertion phase, it is easy to see that they will stay equal after every removal. That is, if the ball is removed from the bin  $k$  then the element is removed from the queue  $k$ . The *rank cost* at a step is the *rank* of the label of the removed ball among all labels still present in all the bins. Thus, to minimize this cost, we would like to always remove the ball of lowest label, but this is obviously unlikely due to the random nature of our process. However, we will show that the rank cost at each step is still well-bounded in expectation. Because of the coupling, this will imply that the rank cost in the  $SMQ$  process is well-bounded in expectation as well, we therefore focus on the analysis of the ranks in the balls into bins process.

Let  $\ell_i(t)$  be the label on top of bin  $i$  at time step  $t$ . Let  $x_i(t) = \ell_i(t)/n$  be the normalized value of this label, and let  $\mu(t) = \frac{1}{n} \sum_{i=1}^n x_i(t)$  be the average normalized label at time step  $t$ . As in [21, 5], we will be analyzing the potential function

$$\Gamma(t) = \sum_{i=1}^n e^{-\alpha(x_i(t) - \mu(t))} + \sum_{i=1}^n e^{\alpha(x_i(t) - \mu(t))}, \quad (2)$$

for a suitable constant  $\alpha$ , which we will define later.

We now overview the analysis of [5], and then proceed to outline the major differences. A key ingredient of this analysis is the  $(1 + \beta)$ -choice random process [21], which is similar to the continuous process we defined above, with the difference that in order to delete an element, with probability  $(1 - \beta)$  we choose a single bin uniformly at random and delete from it, and with probability  $\beta$  we choose two bins uniformly at random and delete from the one which has a ball of lower label on top. (Thus, in expectation, we perform  $(1 + \beta)$  choices at a step.)

In our analysis, we will aim to “loosely couple” the  $SMQ$  process with a variant of the  $(1 + \beta)$ -choice process, with parameters  $\beta = \Omega(\gamma)$  and  $\alpha = \Theta(\beta)$ . As we will see, this coupling will not be exact (unlike the coupling between the discrete and continuous processes above). Yet, we will be able to use the properties of the  $(1 + \beta)$ -choice process to bound the properties of the  $SMQ$  process.

Returning to the  $(1 + \beta)$ -choice process, Lemma 2 in [5] shows the following potential bound, for any step  $t \geq 0$ :

$$\mathbb{E}[\Gamma(t+1) | x_1(t), x_2(t), \dots, x_n(t)] \leq \left(1 - \Omega\left(\frac{\beta^2}{n}\right)\right) \Gamma(t) + poly\left(\frac{1}{\beta}\right). \quad (3)$$

Assuming that Inequality (3) holds, Lemma 3 in [5] proves that for any time step  $t \geq 0$ :

$$\mathbb{E}[\Gamma(t)] \leq O\left(poly\left(\frac{1}{\beta}\right) n\right). \quad (4)$$

Finally, given that (4) holds, Theorems 4 and 5 in [5] show that for  $t \geq 0$ , expected maximum rank of labels on top of bins at time step  $t$  is  $O\left(\frac{n}{\beta}(\log n + \log \frac{1}{\beta})\right)$  and expected average rank is  $O\left(\frac{n}{\beta} \log \frac{1}{\beta}\right)$ .

Relative to this argument, we will aim to show that there exists  $\beta$  such that Equation (3) holds for continuous process. This will imply bounds on the expected average rank and expected maximum rank based on  $\beta$ .

We would like to point out that we provide Lemma and Theorem numbers based on full version of [5].

**Bounding the Potential.** Fix an arbitrary time step  $t$ , and the labels  $x_1(t), x_2(t), \dots, x_n(t)$  on top of the bins. Let  $\Gamma_c(t+1)$  be the potential at time step  $t+1$  if the label is deleted by our continuous process, and let  $\Gamma_\beta(t+1)$  be the potential at time step  $t+1$  if the label is deleted by  $1+\beta$  process. Our goal is to show that there exists  $\beta$  such that

$$\mathbb{E}[\Gamma_c(t+1)|x_1(t), x_2(t), \dots, x_n(t)] \leq \mathbb{E}[\Gamma_\beta(t+1)|x_1(t), x_2(t), \dots, x_n(t)]. \quad (5)$$

We assume that bins are sorted in the increasing order of their top labels. Let  $S_c(i)$  be the probability that we delete a label from one of the first  $i$  counters in our process continuous process, and let  $S_\beta(i)$  be the same probability for the  $(1+\beta)$  process. We can prove the following property:

**Lemma 2.** *If  $\gamma\left(\frac{1}{p_{steal}} - 1\right) \leq \frac{1}{2n}$  and  $\beta = \frac{p_{steal}}{2(1+\gamma)}$ , we get that for any  $1 \leq i \leq n$ ,  $S_c(i) \geq S_\beta(i)$ .*

*Proof.* First, notice that we have that:

$$S_\beta(i) = (1-\beta)\frac{i}{n} + \beta\frac{i^2 + 2i(n-i)}{n^2} = \frac{i}{n} + \beta\frac{i(n-i)}{n^2}.$$

Further, recall that, for each bin  $i$ ,  $1-\gamma \leq \frac{1}{\pi_i n} \leq 1+\gamma$ , for  $\gamma \leq 1/2$ . Here we slightly abuse the notation for  $\pi_i$ , since we assumed that bins are sorted in increasing label order, but this does not change the proof since we will only need to show the lower bound on  $\pi_i$ , which holds for every  $i$ . We have that for any  $i$ ,  $\pi_i \geq \frac{1}{n(1+\gamma)}$ . Let  $P_i = \sum_{j=1}^i \pi_j \geq \frac{i}{n(1+\gamma)}$ . We get that :

$$\begin{aligned} S_c(i) &= P_i + (1-P_i)p_{steal}\frac{i}{n} = P_i\left(1 - p_{steal}\frac{i}{n}\right) + p_{steal}\frac{i}{n} \\ &\geq \frac{i}{(1+\gamma)n}\left(1 - p_{steal}\frac{i}{n}\right) + p_{steal}\frac{i}{n}. \end{aligned}$$

Thus:

$$S_c(i) - S_\beta(i) \geq \frac{i}{(1+\gamma)n} - p_{steal}\frac{i^2}{(1+\gamma)n^2} - (1-p_{steal})\frac{i}{n} - \beta\frac{i(n-i)}{n^2}.$$

Hence, to complete the proof we need to show that

$$\begin{aligned} \beta &\leq \left(\frac{i}{(1+\gamma)n} - p_{steal}\frac{i^2}{(1+\gamma)n^2} - (1-p_{steal})\frac{i}{n}\right)\frac{n^2}{i(n-i)} \\ &= \frac{n}{(n-i)}\left(\frac{1}{1+\gamma} - (1-p_{steal}) - p_{steal}\frac{i}{(1+\gamma)n}\right). \end{aligned} \quad (6)$$

Recall that  $1 \leq i \leq n$  is an integer and for  $i = n$ ,  $S_c(i) = S_\beta(i) = 1$ . Thus, we need to show that (6) holds for  $1 \leq i \leq n-1$ . Next, we can prove that  $\frac{n}{(n-i)}\left(\frac{1}{1+\gamma} - (1-p_{steal}) - p_{steal}\frac{i}{(1+\gamma)n}\right)$  is minimized for  $i = n-1$ . Thus, after plugging  $i = n-1$  in (6), we need to show that

$$\beta \leq n\left(\frac{1}{1+\gamma} - (1-p_{steal}) - p_{steal}\frac{(n-1)}{(1+\gamma)n}\right) = p_{steal}\frac{n}{1+\gamma}\left(\frac{1}{n} - \gamma\left(\frac{1}{p_{steal}} - 1\right)\right).$$

We can now set  $\gamma\left(\frac{1}{p_{steal}} - 1\right) \leq \frac{1}{2n}$  and  $\beta = \frac{p_{steal}}{2(1+\gamma)}$ , and the above inequality holds. This completes the proof of the lemma.  $\square$

We can now show that the potential bound (3) holds. Formally:

**Lemma 3.** Fix any time step  $t$  and labels  $x_1(t), x_2(t), \dots, x_n(t)$ . Let  $\gamma\left(\frac{1}{p_{steal}} - 1\right) \leq \frac{1}{2n}$  and  $\beta = \frac{p_{steal}}{2(1+\gamma)}$ . Also, let  $w_t$  be the random weight which we use to generate new labels for both continuous and  $1 + \beta$  processes. Then:

$$\mathbb{E}[\Gamma_c(t+1) - \Gamma_\beta(t+1)|x_1(t), x_2(t), \dots, x_n(t), w_t] \leq 0. \quad (7)$$

*Proof.* To show the proof of the lemma we use the coupling similar to the one used in [21], Theorem 3.1. We again assume that bins are sorted in the increasing label order. The coupling works as follows: At step  $t$  we pick probability  $0 \leq p < 1$  uniformly at random, for our continuous process we delete label from bin  $i$ , such that  $S_c(i-1) \leq p < S_c(i)$  (we assume that  $S_c(0) = 0$ ) and for  $1 + \beta$  process we delete label from bin  $j$ , if  $S_\beta(j-1) \leq p < S_\beta(j)$ . We set  $x_{c,i}(t+1) = x_i(t) + w_t/n$  and  $x_{\beta,j}(t+1) = x_j(t) + w_t/n$ . Here  $x_{c,i}(t+1)$  and  $x_{\beta,j}(t+1)$  are new normalized labels on top of bins  $i$  and  $j$  for our continuous and  $1 + \beta$  processes correspondingly (The rest of the labels do not change). It is straightforward to verify that coupling is valid, since  $S_c(i) - S_c(i-1)$  is exactly the probability of deleting label from bin  $i$  in our continuous process (the same thing is valid for  $1 + \beta$  process). We would like to note that we are not able to use Theorem 3.1 in [21] directly since it assumes that  $w_t = 1$ . Lemma 2 gives us that  $i \leq j$ , and hence  $x_i(t) \leq x_j(t)$ . Recall that  $\mu(t) = \frac{1}{n} \sum_{k=1}^n x_k(t)$ , for both processes  $\mu(t+1) = \mu(t) + w_t/n^2$ . and for  $k \neq i, j$ ,  $x_k(t+1) = x_k(t)$ , hence to prove that potential at step  $t$  is smaller for continuous process we just need to check at how new labels of bins  $i$  and  $j$  effect potentials.

First we show that

$$e^{\alpha(x_i(t)+w_t/n-w_t/n^2-\mu(t))} + e^{\alpha(x_j(t)-w_t/n^2-\mu(t))} \leq e^{\alpha(x_i(t)-w_t/n^2-\mu(t))} + e^{\alpha(x_j(t)+w_t-w_t/n^2-\mu(t))}.$$

which after diving both sides by  $e^{\alpha(-w_t/n^2-\mu(t))} \geq 0$  is the same as

$$e^{\alpha(x_i(t)+w_t/n)} + e^{\alpha(x_j(t))} \leq e^{\alpha(x_i(t))} + e^{\alpha(x_j(t)+w_t/n)}.$$

After rearranging terms, this can be rewritten as

$$(e^{\alpha w_t/n} - 1)(e^{\alpha(x_j(t))} - e^{\alpha(x_i(t))}) \geq 0.$$

The above inequality holds since  $\alpha \geq 0$ ,  $w_t \geq 0$  and  $x_j(t) \geq x_i(t)$ . Next, we show that

$$e^{-\alpha(x_i(t)+w_t/n-w_t/n^2-\mu(t))} + e^{-\alpha(x_j(t)-w_t/n^2-\mu(t))} \leq e^{-\alpha(x_i(t)-w_t/n^2-\mu(t))} + e^{-\alpha(x_j(t)+w_t-w_t/n^2-\mu(t))}.$$

which is the same as

$$e^{-\alpha(x_i(t)+w_t/n)} + e^{-\alpha(x_j(t))} \leq e^{-\alpha(x_i(t))} + e^{-\alpha(x_j(t)+w_t/n)}.$$

Rearranging terms, this can be rewritten as

$$(e^{-\alpha w_t/n} - 1)(e^{-\alpha(x_j(t))} - e^{-\alpha(x_i(t))}) \geq 0.$$

The above inequality clearly holds since  $\alpha \geq 0$ ,  $w_t \geq 0$  and  $x_j(t) \geq x_i(t)$  (in this case, both terms are negative).  $\square$

**Proof of Theorem 1.** Finally, we can prove our main result. First, fix normalized labels on the top of the bins:  $x_1(t), x_2(t), \dots, x_n(t)$ . Lemma 2 in full version of [5] shows that for any step  $t \geq 0$ :

$$\mathbb{E}[\Gamma_\beta(t+1)|x_1(t), x_2(t), \dots, x_n(t)] \leq \left(1 - \Omega\left(\frac{\beta^2}{n}\right)\right)\Gamma(t) + poly\left(\frac{1}{\beta}\right). \quad (8)$$

and Lemma 3 gives us that for  $\beta = \frac{p_{steal}}{2(1+\gamma)}$

$$\mathbb{E}[\Gamma_c(t+1) - \Gamma_\beta(t+1)|x_1(t), x_2(t), \dots, x_n(t), w_t] \leq 0.$$

We remove conditioning on  $w_t$  and slightly abuse the notation in  $1 + \beta$  process: we assume that  $w_t$  is  $Exp(\pi_i)$  if continuous process deletes from bin  $i$ , this slightly changes  $1 + \beta$  process (which might delete from different bin), but proof of (8) in [5] will still be correct since it only uses that  $w_t = Exp(\pi_i)$ , (To be more precise, the proof uses expectation and moment generating function of exponential random variable) for  $1 - \gamma \leq \frac{1}{\pi_i n} \leq 1 + \gamma$ , and  $\gamma \leq 1/2$  and does not make any assumptions about  $i$ . Hence we get that

$$\mathbb{E}[\Gamma_c(t+1) - \Gamma_\beta(t+1)|x_1(t), x_2(t), \dots, x_n(t)] \leq 0.$$

and this means that

$$\mathbb{E}[\Gamma_c(t+1)|x_1(t), x_2(t), \dots, x_n(t)] \leq \left(1 - \Omega\left(\frac{\beta^2}{n}\right)\right)\Gamma(t) + \text{poly}\left(\frac{1}{\beta}\right).$$

Lemma 3 in the full version of [5] proves that for any time step  $t \geq 0$ :

$$\mathbb{E}[\Gamma_c(t)] \leq O\left(\text{poly}\left(\frac{1}{\beta}\right)n\right). \quad (9)$$

Finally, we can use Theorems 4 and 5 in the full version of [5] to show that for the continuous process and time  $t \geq 0$ , the expected maximum rank of labels on top of bins at time step  $t$  is  $O\left(\frac{n}{\beta}(\log n + \log \frac{1}{\beta})\right)$  and expected average rank is  $O\left(\frac{n}{\beta} \log \frac{1}{\beta}\right)$ . This is possible since these theorems only use the upper bound on  $\mathbb{E}[\Gamma(t)]$ , regardless of which process we used to derive this bound). Plugging in  $\beta = \frac{p_{\text{steal}}}{2(1+\gamma)}$  in the above rank bounds and using Lemma 1 completes the proof of the theorem for  $B = 1$ . We proceed by specifying what will change in the proof if  $B > 1$ . Lemma 1 holds as before. In the continuous process we delete  $B$  labels instead of just one. This means that if continuous process deletes  $B$  labels from bin  $i$  at step  $t$ . We have that  $\ell_i(t+1) = \ell_i(t) + \sum_{k=1}^B \text{Exp}(\pi_i)$  (Recall that  $\ell_i(t)$  is a label on top of bin  $i$  at step  $t$ ). Hence, we use a normalized label:  $x_i(t) = \frac{\ell_i(t)}{Bn}$ . Consider random variable  $\sum_{k=1}^B \frac{\text{Exp}(\pi_i)}{B}$  (this is by how much normalized label on top of bin  $i$  increases, if we ignore factor of  $1/n$ ). We have that  $\mathbb{E}\left[\sum_{k=1}^B \frac{\text{Exp}(\pi_i)}{B}\right] = \mathbb{E}\left[\text{Exp}(\pi_i)\right]$  Also, by convexity of exponential function and Jensen's inequality we have that the moment generating function of  $\sum_{k=1}^B \text{Exp}(\pi_i)/B$  is upper bounded by moment generating function of  $\text{Exp}(\pi_i)$  (This is important since the proof of (8) in [5] uses moment generating function). This means that we can apply all the steps of the proof exactly as in the case of  $B = 1$ . The only difference will be last step, where we scale back by  $1/nB$  instead of  $1/n$ . Hence bounds on expected maximum rank and expected average rank become  $O\left(\frac{nB}{\beta}(\log n + \log \frac{1}{\beta})\right)$  and  $O\left(\frac{nB}{\beta} \log \frac{1}{\beta}\right)$ .

## B Tuning OBIM and PMOD Schedulers

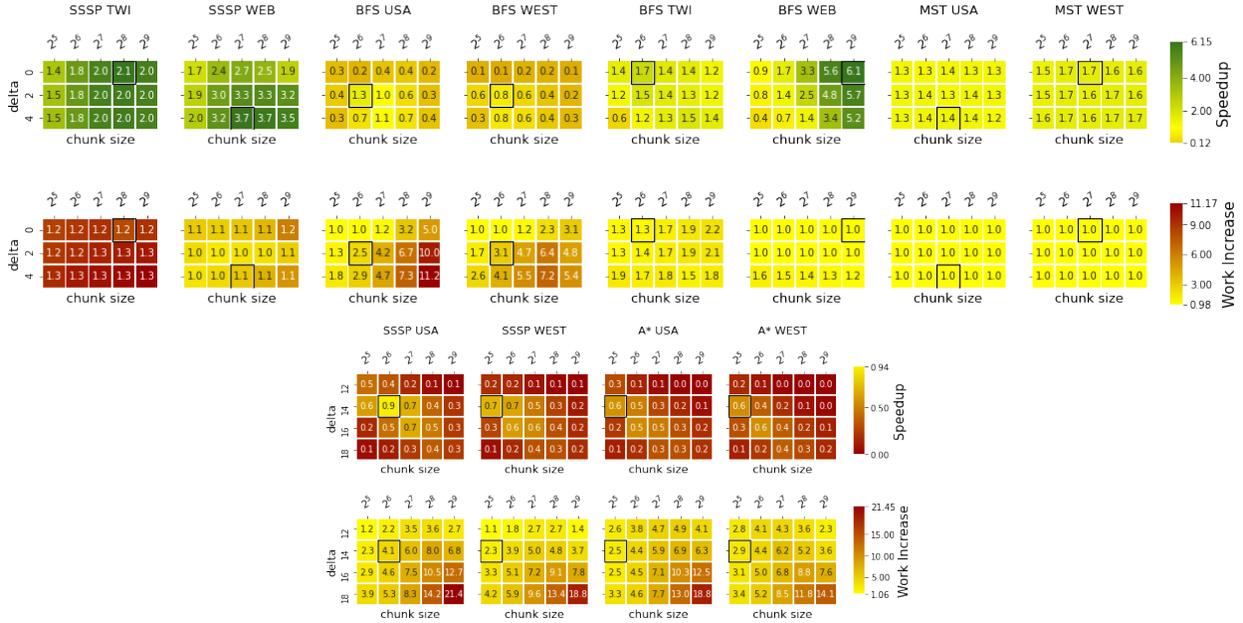


Figure 3: Ablation of  $\Delta$  and `CHUNK_SIZE` parameters for OBIM. Experiments execute on 256 threads on the AMD platform. The baseline is the classic Multi-Queue on 256 threads with  $C = 4$ . The fastest configuration for each benchmark is highlighted with a black border. Best viewed in color.

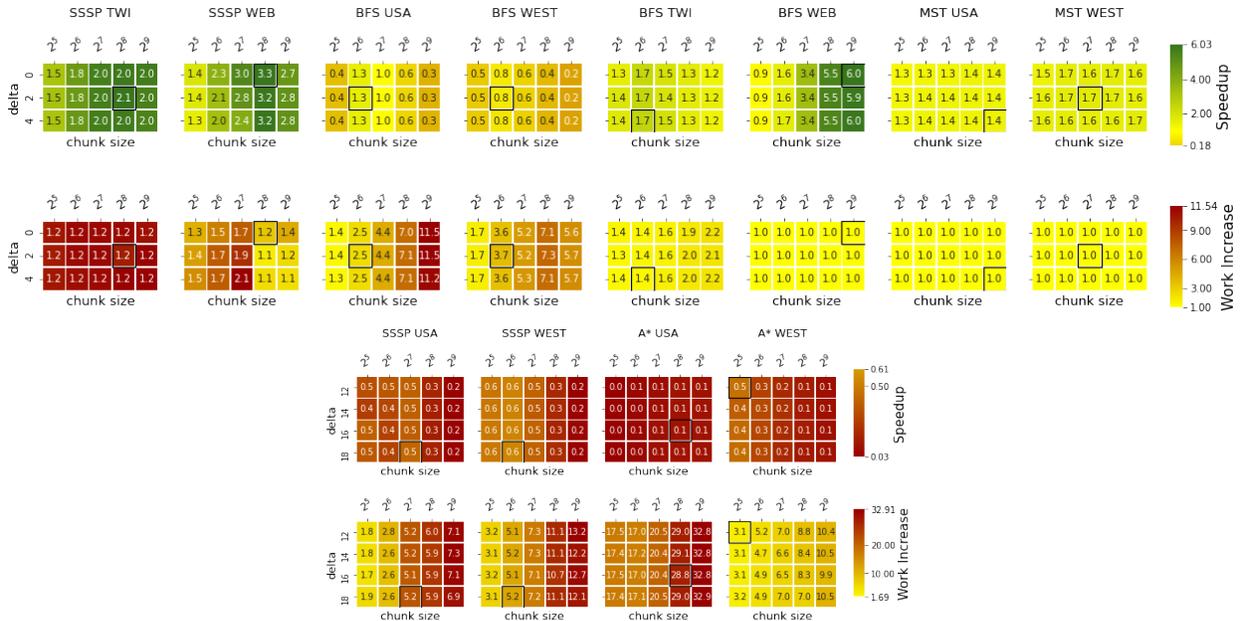


Figure 4: Ablation of  $\Delta$  and `CHUNK_SIZE` parameters for PMOD. Experiments execute on 256 threads on the AMD platform. The baseline is the classic Multi-Queue on 256 threads with  $C = 4$ . The fastest configuration for each benchmark is highlighted with a black border. Best viewed in color.

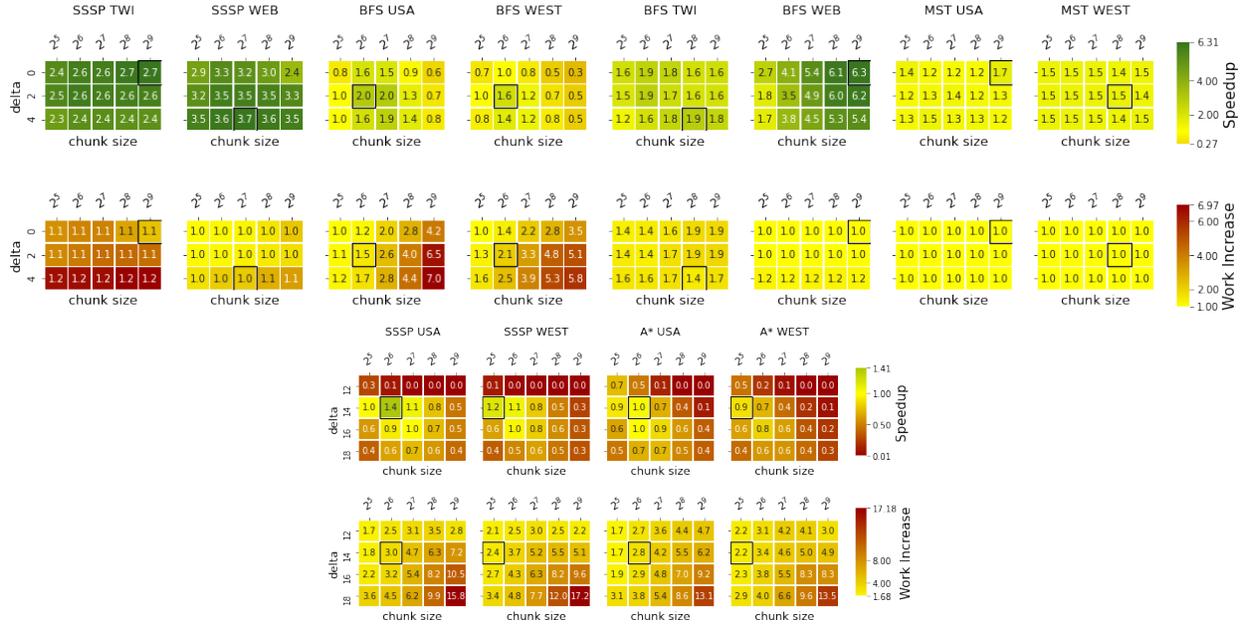


Figure 5: Ablation of  $\Delta$  and  $\text{CHUNK\_SIZE}$  parameters for OBIM. Experiments execute on 128 threads on the **Intel** platform. The baseline is the classic Multi-Queue on 128 threads with  $C = 4$ . The fastest configuration for each benchmark is highlighted with a black border. Best viewed in color.

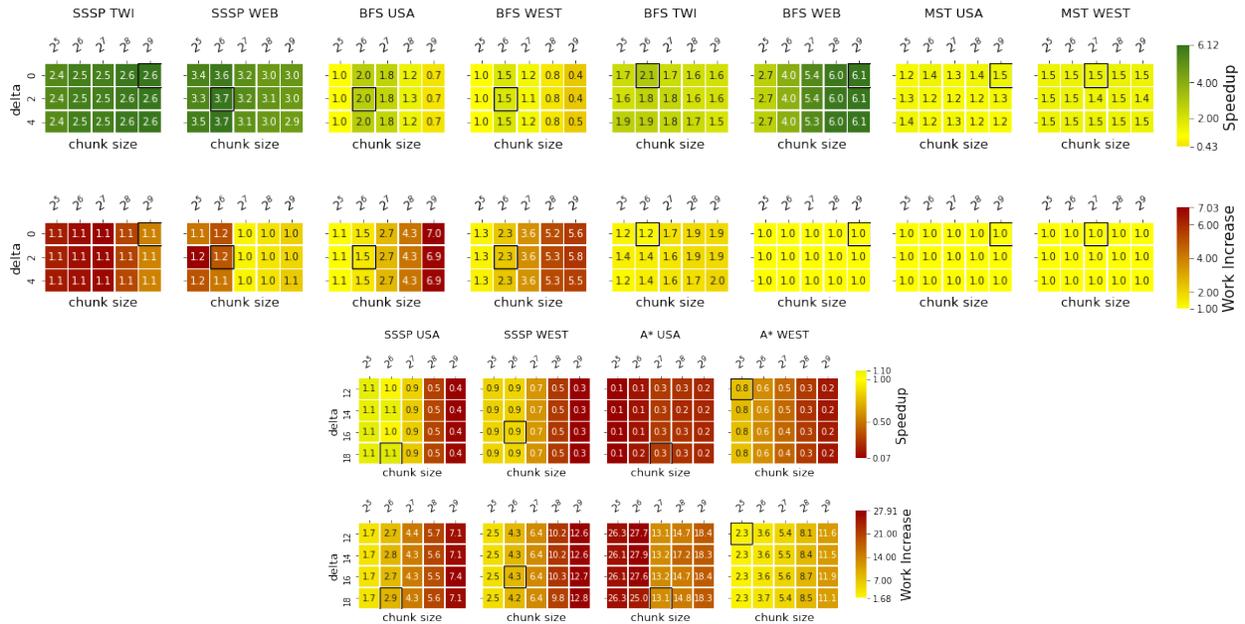


Figure 6: Ablation of  $\Delta$  and  $\text{CHUNK\_SIZE}$  parameters for PMOD. Experiments execute on 128 threads on the **Intel** platform. The baseline is the classic Multi-Queue on 128 threads with  $C = 4$ . The fastest configuration for each benchmark is highlighted with a black border. Best viewed in color.

## C Classic Multi-Queue Optimizations

Usually, Multi-Queues use  $C \times T$  sequential queues under the hood, where  $T$  is the number of threads and  $C$  is some constant between 2 and 8. We consider different  $C$  values to find the best one, and present the corresponding experimental data in Tables 2–3.

After that, we evaluate the *task batching* and *temporal locality* optimizations in Subsection 2.1. In short, task batching reduces the ratio between synchronization cost and task execution time by retrieving multiple tasks from the same queue at once in `delete()` and buffering them in `insert(..)`. The temporal locality optimization reduces the cache coherence overhead by using the same queue for a sequence of `delete()` or `insert(..)` operations, changing the “temporally local” queue with a constant probability. We evaluate all the four combinations of these optimizations, and present the corresponding results in Figures 7–14. In addition, we compare the optimally configured combinations — the results are presented in Figures 15–16. All the experiments are executed on both AMD and Intel platforms.

	2	3	4	5	6	7	8
<b>BFS USA</b>	17.18	21.33	23.41	25.76	27.28	<b>28.27</b>	27.44
<b>BFS WEST</b>	17.64	20.31	20.69	22.05	23.10	23.65	<b>23.77</b>
<b>BFS TWI</b>	<b>30.20</b>	30.06	29.28	27.81	29.09	28.98	28.29
<b>BFS WEB</b>	15.95	16.41	17.65	18.03	18.06	<b>18.46</b>	18.42
<b>SSSP USA</b>	17.83	<b>18.26</b>	17.53	18.10	17.35	16.95	16.34
<b>SSSP WEST</b>	13.18	<b>13.80</b>	13.52	12.67	11.40	11.74	11.29
<b>SSSP TWI</b>	<b>27.01</b>	26.86	26.51	24.98	24.14	25.11	25.03
<b>SSSP WEB</b>	26.70	26.70	27.30	26.91	28.93	<b>29.15</b>	<b>29.15</b>
<b>MST USA</b>	13.04	11.97	12.55	11.46	12.71	<b>13.25</b>	12.87
<b>MST WEST</b>	7.63	7.38	7.73	7.73	<b>7.87</b>	7.22	7.40
<b>A* USA</b>	23.47	24.74	<b>26.27</b>	26.12	25.57	24.79	23.96
<b>A* WEST</b>	18.01	<b>19.52</b>	18.78	18.31	17.57	16.64	16.06

Table 2: Speedup of the classic Multi-Queue with various  $C$  executed on 256 threads on the **AMD** platform. The baseline is sequential priority queue execution on a single thread. The best speedups are highlighted with **red**.

	2	3	4	5	6	7	8
<b>BFS USA</b>	9.45	11.65	12.70	13.49	13.74	<b>13.77</b>	<b>13.77</b>
<b>BFS WEST</b>	9.35	11.00	12.19	12.76	13.01	12.95	<b>13.60</b>
<b>BFS TWI</b>	29.00	29.04	28.62	26.81	<b>29.12</b>	28.99	27.91
<b>BFS WEB</b>	12.32	13.05	13.52	13.94	13.91	14.88	<b>15.03</b>
<b>SSSP USA</b>	10.40	12.68	13.49	<b>13.53</b>	13.35	13.04	13.19
<b>SSSP WEST</b>	9.45	9.85	<b>10.96</b>	10.83	10.53	10.04	9.74
<b>SSSP TWI</b>	27.30	26.98	<b>27.64</b>	27.30	26.36	26.30	26.90
<b>SSSP WEB</b>	23.65	24.58	25.17	25.22	23.79	26.13	<b>26.51</b>
<b>MST USA</b>	4.50	4.73	4.66	4.44	4.58	4.73	<b>4.86</b>
<b>MST WEST</b>	4.78	4.66	<b>4.80</b>	4.46	4.59	4.33	4.75
<b>A* USA</b>	14.64	17.42	17.73	18.31	<b>18.71</b>	18.58	18.44
<b>A* WEST</b>	12.29	14.10	14.55	<b>14.59</b>	14.25	14.01	13.60

Table 3: Speedup of the classic Multi-Queue with various  $C$  executed on 128 threads on the **Intel** platform. The baseline is sequential priority queue execution on a single thread. The best speedups are highlighted with **red**.

### C.1 Classic Multi-Queue Optimizations on AMD: insert=Temporal Locality, delete=Temporal Locality

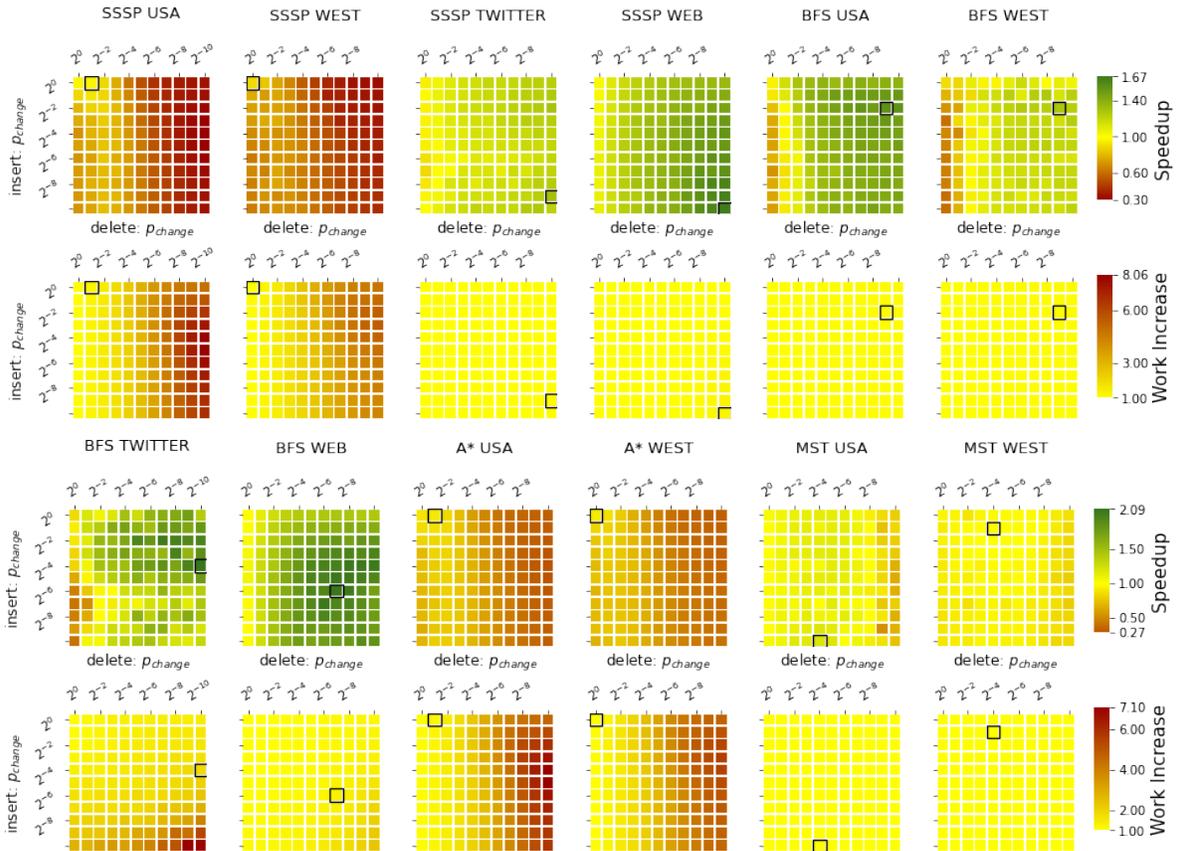


Figure 7: Ablation of queue change probabilities (Temporal Locality) for both `insert(.)` and `delete()`. Experiments execute on 256 threads on the **AMD** platform. The baseline is the classic Multi-Queue on 256 threads with  $C = 4$ . The fastest configuration for each benchmark is highlighted with a black border and listed in Table 4.

	<b>SSSP USA</b>	<b>SSSP WEST</b>	<b>SSSP TWITTER</b>	<b>SSSP WEB</b>	<b>BFS USA</b>	<b>BFS WEST</b>
$p_{insert}$	1/1	1/1	1/512	1/1024	1/4	1/4
$p_{delete}$	1/2	1/1	1/1024	1/1024	1/512	1/512
Speedup	0.98	0.91	1.32	1.67	1.56	1.29
Work Increase	1.24	1.04	1.07	1.31	1.07	1.07
	<b>BFS TWITTER</b>	<b>BFS WEB</b>	<b>A* USA</b>	<b>A* WEST</b>	<b>MST USA</b>	<b>MST WEST</b>
$p_{insert}$	1/16	1/64	1/1	1/1	1/1024	1/2
$p_{delete}$	1/1024	1/128	1/2	1/1	1/16	1/16
Speedup	1.19	2.09	0.91	0.92	1.19	1.01
Work Increase	1.03	1.01	1.19	1.03	1.00	1.00

Table 4: The optimal parameters for Multi-Queue with the *temporal locality* optimization for both `insert(.)` and `delete()` obtained on the **AMD** platform. Based on Figure 7. For each benchmark, the best parameters are presented with the speedup and work increase.

## C.2 Classic Multi-Queue Optimizations on Intel: insert=Temporal Locality, delete=Temporal Locality

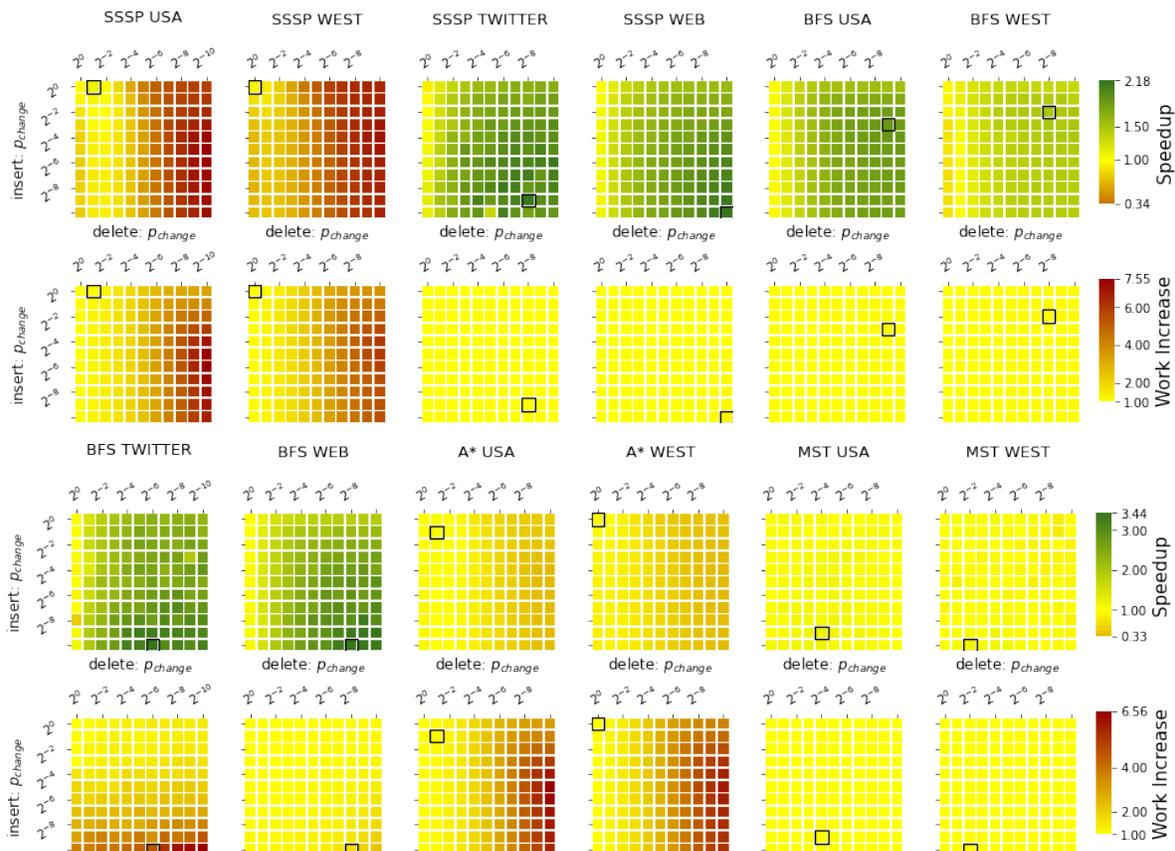


Figure 8: Ablation of queue change probabilities (Temporal Locality) for both `insert(.)` and `delete()`. Experiments execute on 128 threads on the **Intel** platform. The baseline is the classic Multi-Queue on 128 threads with  $C = 4$ . The fastest configuration for each benchmark is highlighted with a black border and listed in Table 5.

	SSSP USA	SSSP WEST	SSSP TWITTER	SSSP WEB	BFS USA	BFS WEST
$p_{insert}$	1/1	1/1	1/512	1/1024	1/8	1/4
$p_{delete}$	1/2	1/1	1/256	1/1024	1/512	1/256
Speedup	1.05	0.97	1.73	2.18	1.88	1.50
Work Increase	1.15	1.04	1.01	1.15	1.06	1.07
	BFS TWITTER	BFS WEB	A* USA	A* WEST	MST USA	MST WEST
$p_{insert}$	1/1024	1/1024	1/2	1/1	1/512	1/1024
$p_{delete}$	1/64	1/256	1/2	1/1	1/16	1/4
Speedup	1.60	3.44	1.06	0.98	1.20	1.16
Work Increase	1.09	1.02	1.11	1.02	1.00	1.01

Table 5: The optimal parameters for Multi-Queue with the *temporal locality* optimization for both `insert(.)` and `delete()` obtained on the **Intel** platform. Based on Figure 8. For each benchmark, the best parameters are presented with the speedup and work increase.

### C.3 Classic Multi-Queue Optimizations on AMD: insert=Temporal Locality, delete=Task Batching

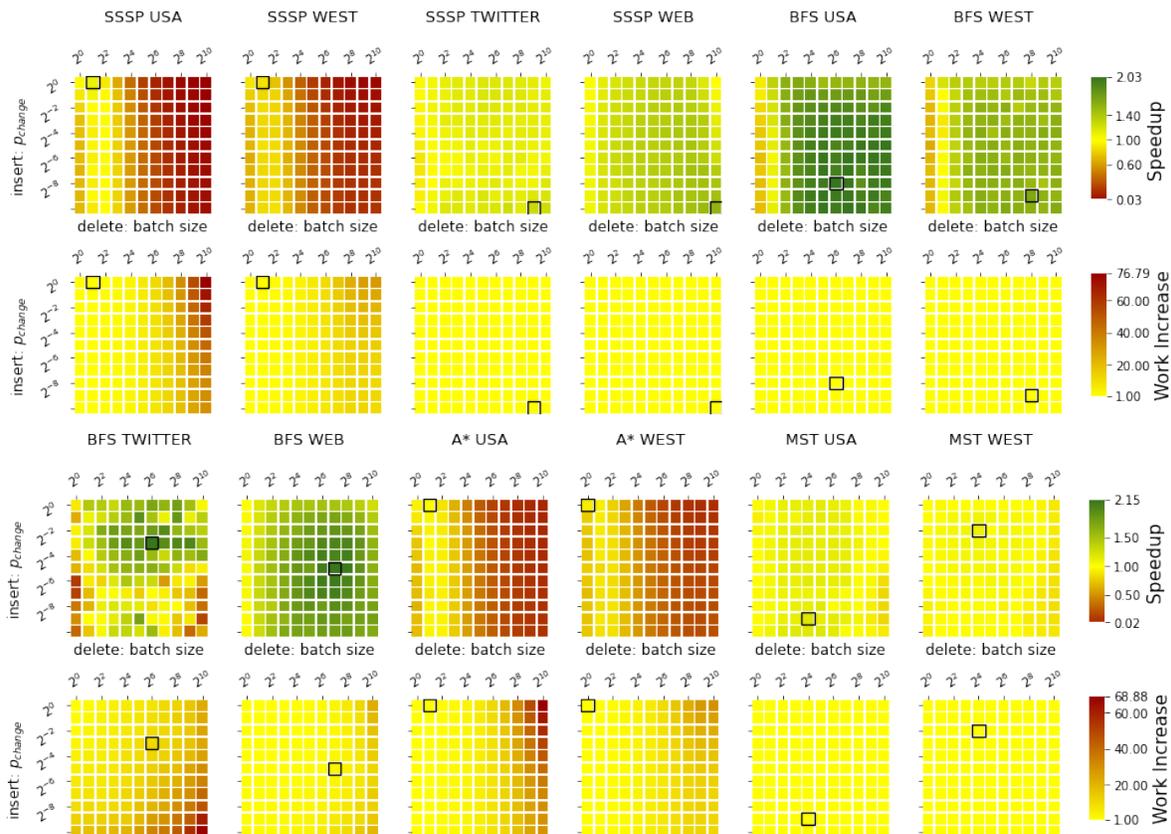


Figure 9: Ablation of queue change probability (Temporal Locality) for `insert(.)` and batch size for `delete()`. Experiments execute on 256 threads on the **AMD** platform. The baseline is the classic Multi-Queue on 256 threads with  $C = 4$ . The fastest configuration for each benchmark is highlighted with a black border and listed in Table 6.

	<b>SSSP USA</b>	<b>SSSP WEST</b>	<b>SSSP TWITTER</b>	<b>SSSP WEB</b>	<b>BFS USA</b>	<b>BFS WEST</b>
$p_{insert}$	1/1	1/1	1/1024	1/1024	1/256	1/512
$BATCH_{delete}$	2	2	512	1024	64	256
Speedup	1.09	0.90	1.25	1.53	2.03	1.65
Work Increase	1.28	1.35	1.10	1.40	1.08	1.08
	<b>BFS TWITTER</b>	<b>BFS WEB</b>	<b>A* USA</b>	<b>A* WEST</b>	<b>MST USA</b>	<b>MST WEST</b>
$p_{insert}$	1/8	1/32	1/1	1/1	1/512	1/4
$BATCH_{delete}$	64	128	2	1	16	16
Speedup	1.15	2.15	1.00	0.93	1.20	1.09
Work Increase	1.05	1.01	1.22	1.03	1.00	1.00

Table 6: The optimal parameters for Multi-Queue with the *temporal locality* optimization for `insert(.)` and *task batching* for `delete()`, obtained on the **AMD** platform. Based on Figure 9. For each benchmark, the best parameters are presented with the speedup and work increase.

## C.4 Classic Multi-Queue Optimizations on Intel: insert=Temporal Locality, delete=Batching

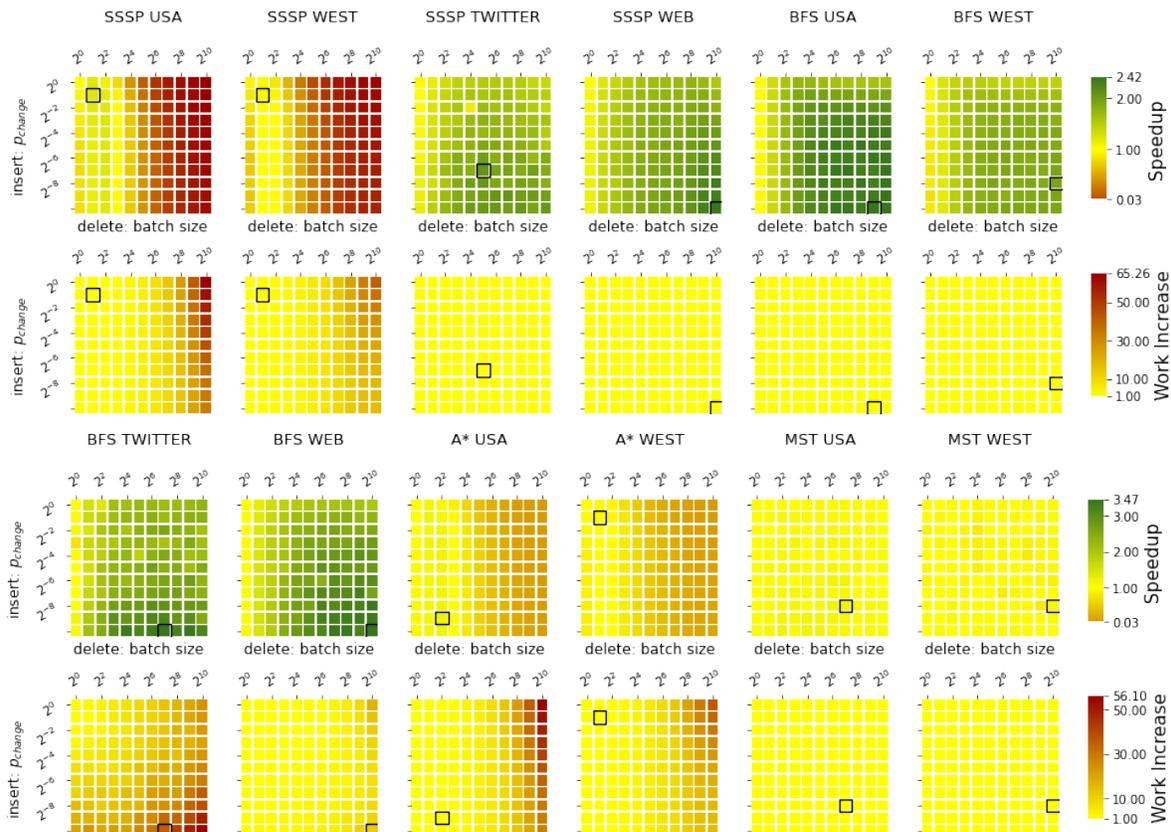


Figure 10: Ablation of queue change probability (Temporal Locality) for `insert(.)` and batch size for `delete()`. Experiments execute on 128 threads on the **Intel** platform. The baseline is the classic Multi-Queue on 128 threads with  $C = 4$ . The fastest configuration for each benchmark is highlighted with a black border and listed in Table 7.

	<b>SSSP USA</b>	<b>SSSP WEST</b>	<b>SSSP TWITTER</b>	<b>SSSP WEB</b>	<b>BFS USA</b>	<b>BFS WEST</b>
$p_{insert}$	1/2	1/2	1/128	1/1024	1/1024	1/256
$BATCH_{delete}$	2	2	32	1024	512	1024
Speedup	1.16	1.03	1.76	2.15	2.42	1.93
Work Increase	1.19	1.30	1.02	1.22	1.07	1.08
	<b>BFS TWITTER</b>	<b>BFS WEB</b>	<b>A* USA</b>	<b>A* WEST</b>	<b>MST USA</b>	<b>MST WEST</b>
$p_{insert}$	1/1024	1/1024	1/512	1/2	1/256	1/256
$BATCH_{delete}$	128	1024	4	2	128	1024
Speedup	1.64	3.47	1.18	1.04	1.20	1.12
Work Increase	1.12	1.04	1.34	1.21	1.00	1.00

Table 7: The optimal parameters for Multi-Queue with the *temporal locality* optimization for `insert(.)` and *task batching* for `delete()`, obtained on **Intel**. Based on Figure 10. For each benchmark, the best parameters are presented with the speedup and work increase.

## C.5 Classic Multi-Queue Optimizations on AMD: insert=Batching, delete=Temporal Locality

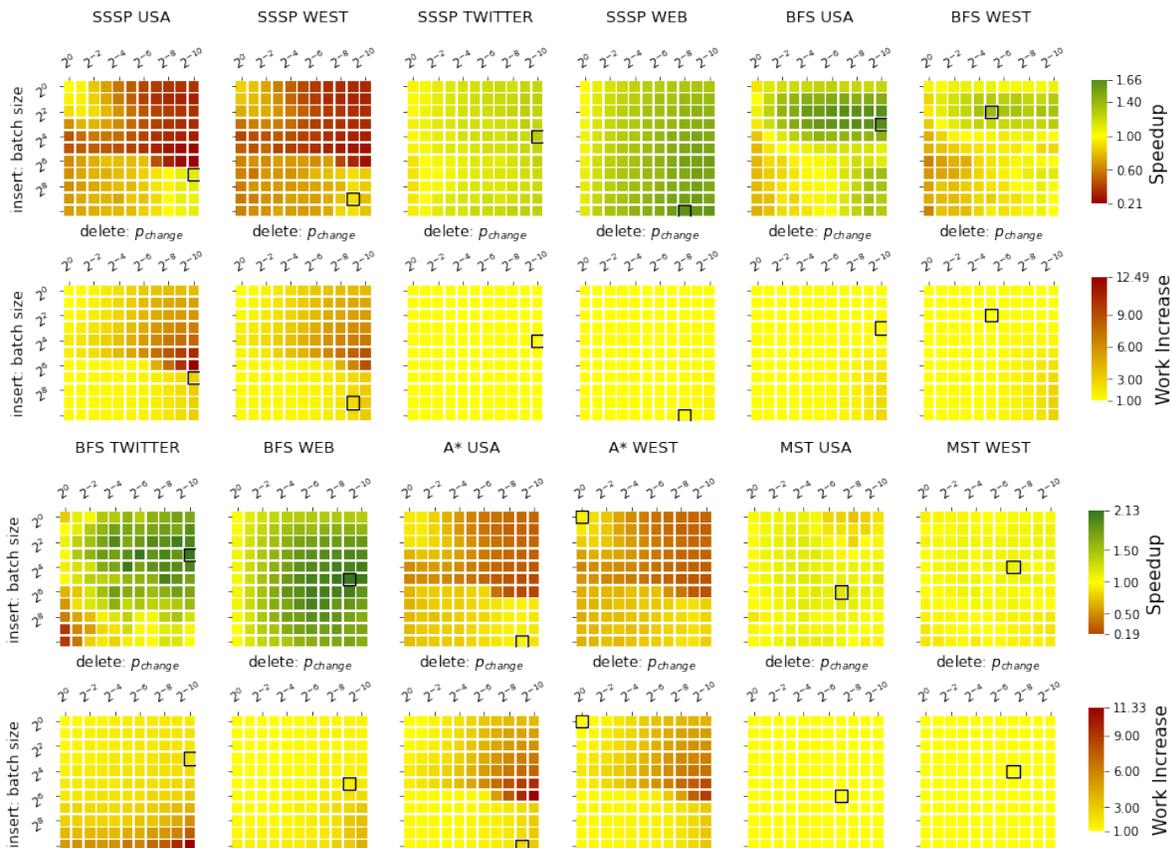


Figure 11: Ablation of batch size ( $B$ ) for  $\text{insert}(\dots)$  and queue change probability ( $TL$ ) for  $\text{delete}()$ . Experiments execute on 256 threads on the **AMD** platform. The baseline is the classic Multi-Queue on 256 threads with  $C = 4$ . The fastest configuration for each benchmark is highlighted with a black border. Best viewed in color.

	SSSP USA	SSSP WEST	SSSP TWITTER	SSSP WEB	BFS USA	BFS WEST
$BATCH_{\text{insert}}$	128	512	16	1024	8	4
$p_{\text{delete}}$	1/1024	1/512	1/1024	1/256	1/1024	1/32
Speedup	1.11	0.91	1.27	1.64	1.66	1.38
Work Increase	3.06	2.60	1.02	1.13	1.08	1.07
	BFS TWITTER	BFS WEB	A* USA	A* WEST	MST USA	MST WEST
$BATCH_{\text{insert}}$	8	32	1024	1	64	16
$p_{\text{delete}}$	1/1024	1/512	1/512	1/1	1/128	1/128
Speedup	1.18	2.13	0.95	0.94	1.19	1.10
Work Increase	1.02	1.02	2.38	1.00	1.00	1.00

Table 8: The optimal parameters for Multi-Queue with the *task batching* optimization for  $\text{insert}(\dots)$  and *temporal locality* for  $\text{delete}()$ , obtained on the **AMD** platform. Based on Figure 11. For each benchmark, the best parameters are presented with the speedup and work increase.

## C.6 Classic Multi-Queue Optimizations on Intel: insert=Batching, delete=Temporal Locality

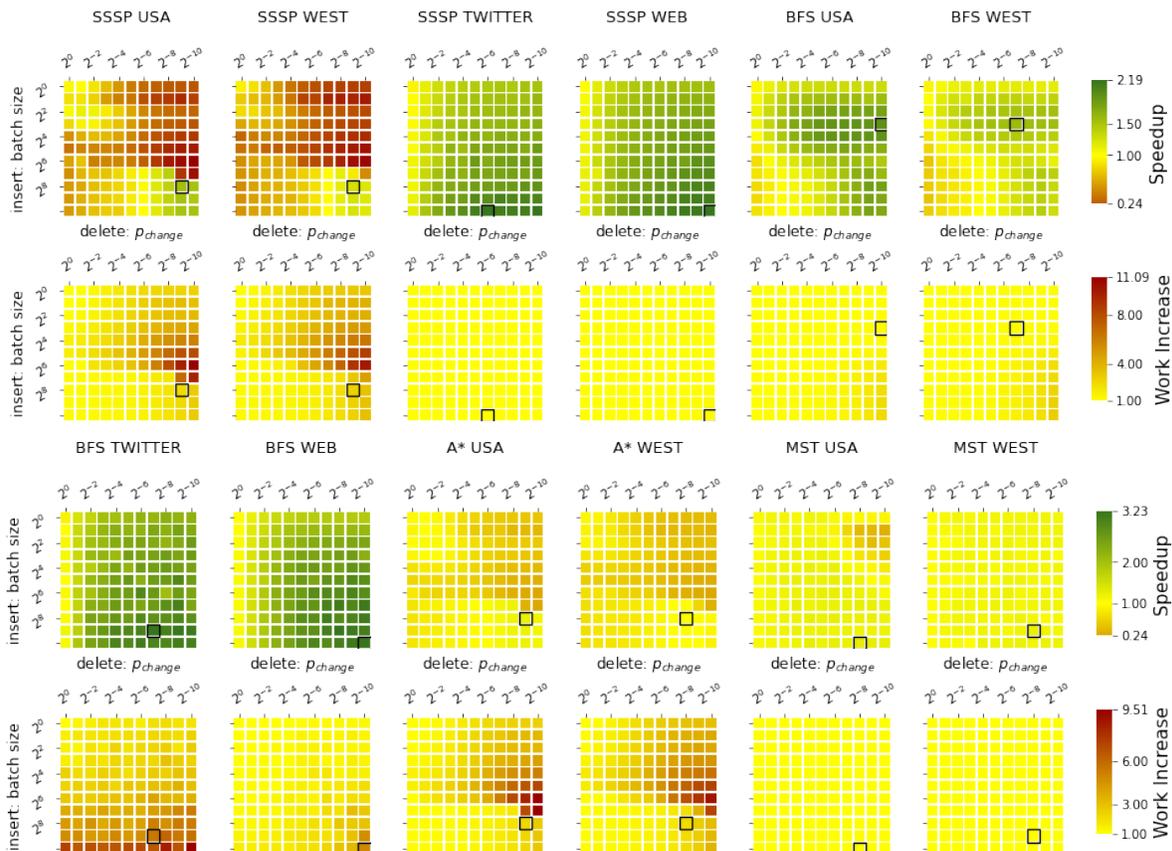


Figure 12: Intel. Ablation of batch size ( $B$ ) for  $\text{insert}(\dots)$  and queue change probability ( $TL$ ) for  $\text{delete}(\dots)$ . Experiments execute on 128 threads. The baseline is the classic Multi-Queue on 128 threads with  $C = 4$ . The fastest configuration for each benchmark is highlighted with a black border. Best viewed in color.

	SSSP USA	SSSP WEST	SSSP TWITTER	SSSP WEB	BFS USA	BFS WEST
$BATCH_{insert}$	256	256	1024	1024	8	8
$p_{delete}$	1/512	1/512	1/64	1/1024	1/1024	1/128
Speedup	1.34	1.16	2.00	2.19	1.95	1.58
Work Increase	2.47	2.80	1.02	1.21	1.08	1.09
	BFS TWITTER	BFS WEB	A* USA	A* WEST	MST USA	MST WEST
$BATCH_{insert}$	512	1024	256	256	1024	512
$p_{delete}$	1/128	1/1024	1/512	1/256	1/256	1/256
Speedup	1.60	3.23	1.22	1.03	1.27	1.24
Work Increase	1.05	1.05	2.28	2.13	1.00	1.00

Table 9: The optimal parameters for Multi-Queue with the *task batching* optimization for  $\text{insert}(\dots)$  and *emph*temporal locality for  $\text{delete}(\dots)$ , obtained on the **Intel** platform. Based on Figure 12. For each benchmark, the best parameters are presented with the speedup and work increase.

## C.7 Classic Multi-Queue Optimizations on AMD: insert=Batching, delete=Batching

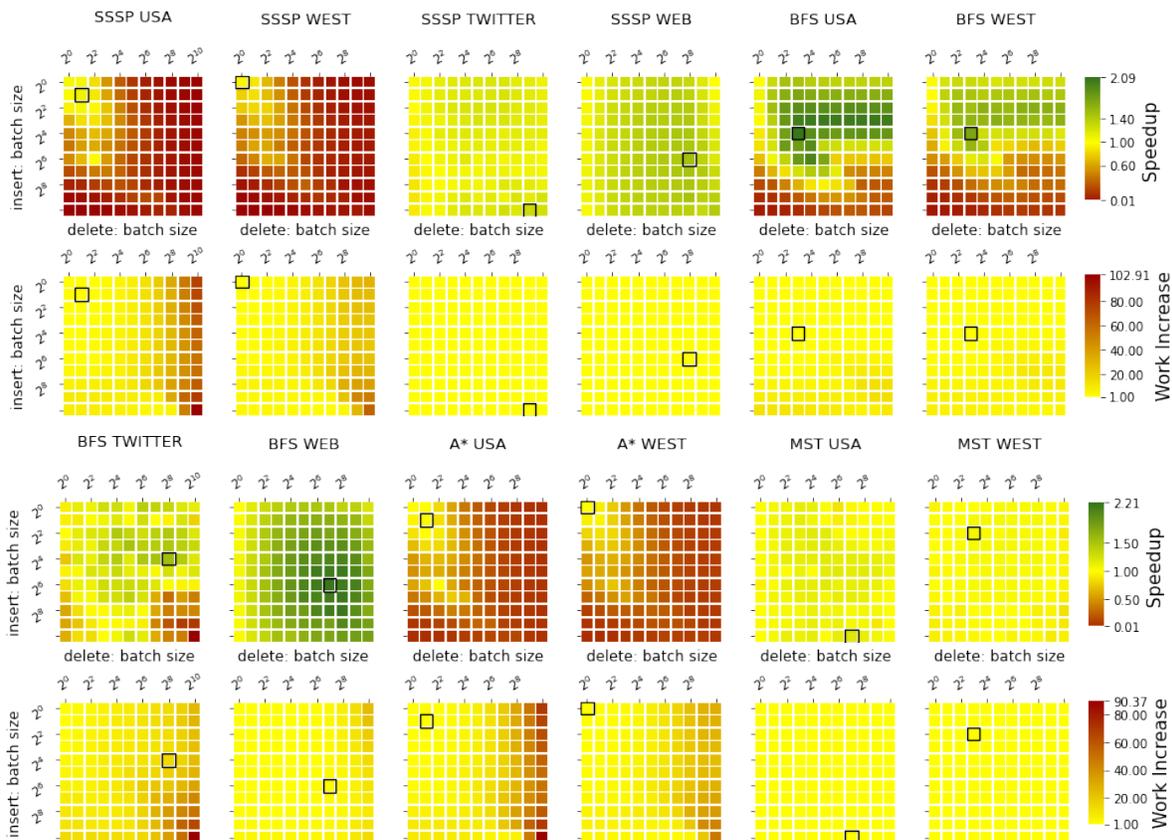


Figure 13: Ablation of batch sizes (B) for `insert(.)` and `delete()`. Experiments execute on 256 threads on the **AMD** platform. The baseline is the classic Multi-Queue on 256 threads with  $C = 4$ . The fastest configuration for each benchmark is highlighted with a black border. Best viewed in color.

	<b>SSSP USA</b>	<b>SSSP WEST</b>	<b>SSSP TWITTER</b>	<b>SSSP WEB</b>	<b>BFS USA</b>	<b>BFS WEST</b>
$BATCH_{insert}$	2	1	1024	64	16	16
$BATCH_{delete}$	2	1	512	256	8	8
Speedup	1.08	0.97	1.25	1.44	2.09	1.71
Work Increase	1.33	1.00	1.12	1.15	1.11	1.29
	<b>BFS TWITTER</b>	<b>BFS WEB</b>	<b>A* USA</b>	<b>A* WEST</b>	<b>MST USA</b>	<b>MST WEST</b>
$BATCH_{insert}$	16	64	2	1	1024	4
$BATCH_{delete}$	256	128	2	1	128	8
Speedup	1.15	2.21	1.02	1.00	1.19	1.07
Work Increase	1.06	1.01	1.27	1.00	1.00	1.00

Table 10: The optimal parameters for Multi-Queue with the *task batching* optimization for both `insert(.)` and `delete()`, obtained on the **AMD** platform. Based on Figure 13. For each benchmark, the best parameters are presented with the speedup and work increase.

## C.8 Classic Multi-Queue Optimizations on Intel: insert=Batching, delete=Batching

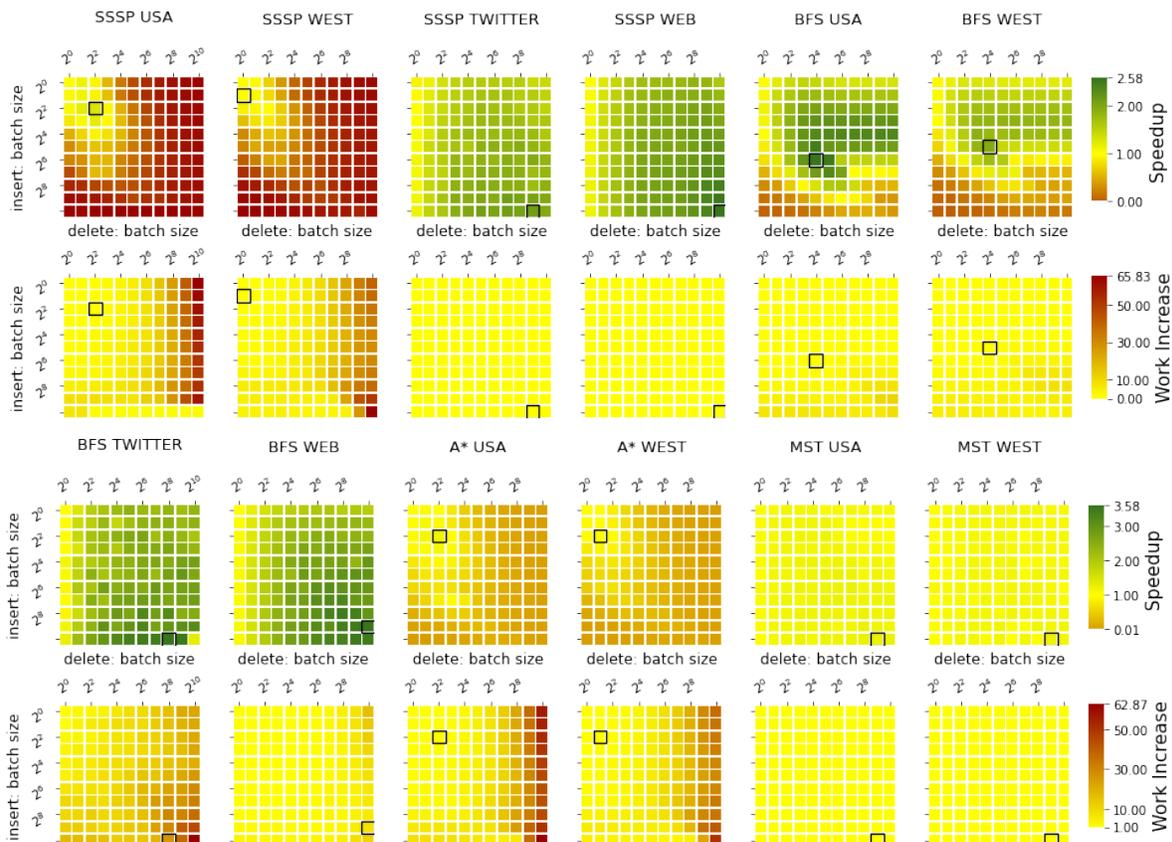


Figure 14: Ablation of batch sizes (B) for insert(.) and delete(). Experiments execute on 128 threads on **Intel**. The baseline is the classic Multi-Queue on 128 threads with  $C = 4$ . The fastest configuration for each benchmark is highlighted with a black border. Best viewed in color.

	<b>SSSP USA</b>	<b>SSSP WEST</b>	<b>SSSP TWITTER</b>	<b>SSSP WEB</b>	<b>BFS USA</b>	<b>BFS WEST</b>
BATCH <sub>insert</sub>	4	2	1024	1024	64	32
BATCH <sub>delete</sub>	4	1	512	1024	16	16
Speedup	1.19	1.00	1.71	2.07	2.58	1.96
Work Increase	1.55	1.08	1.09	1.23	1.22	1.30
	<b>BFS TWITTER</b>	<b>BFS WEB</b>	<b>A* USA</b>	<b>A* WEST</b>	<b>MST USA</b>	<b>MST WEST</b>
BATCH <sub>insert</sub>	1024	512	4	4	1024	1024
BATCH <sub>delete</sub>	256	1024	4	2	512	512
Speedup	1.63	3.58	1.19	1.00	1.26	1.23
Work Increase	1.08	1.02	1.40	1.33	1.00	1.00

Table 11: The optimal parameters for Multi-Queue with the *task batching* optimization for both insert(.) and delete(), obtained on **Intel**. Based on Figure 14. For each benchmark, the best parameters are presented with the speedup and work increase.

### C.9 Comparison of Task Batching and Temporal Locality Optimizations

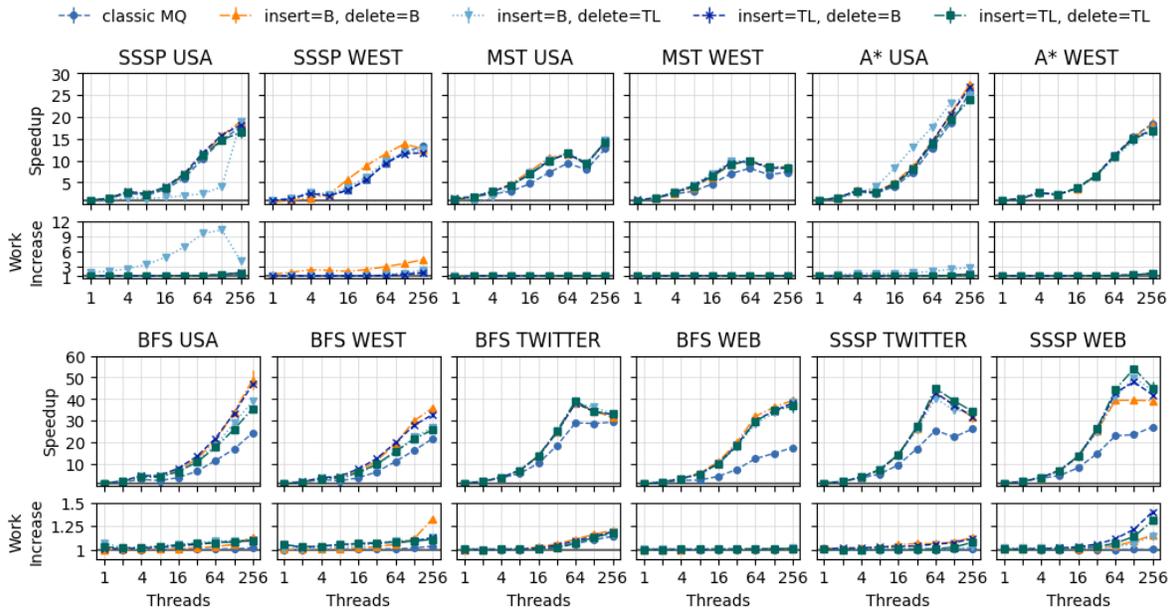


Figure 15: A comparison of task batching (B) and temporal locality (TL) optimizations on the classic Multi-Queue on the AMD platform.

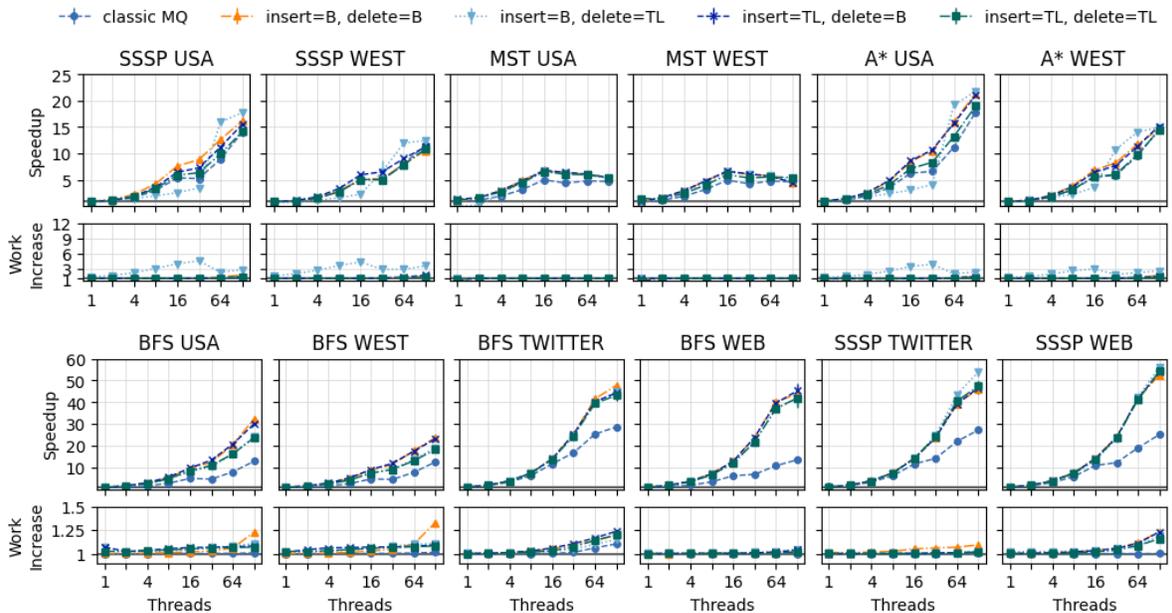


Figure 16: Intel. A comparison of task batching (B) and temporal locality (TL) optimizations on the classic Multi-Queue.

## D Stealing Multi-Queue Implementation Details

This section presents the experimental data for the proposed Stealing Multi-Queue (SMQ) algorithm. Figures 17–18 and Tables 12–13 show results for the version with sequential  $d$ -ary heaps and stealing buffers, while Figures 19–20 and Tables 14–15 show results for the Skip-List based version.

### D.1 SMQ via D-Ary Heaps on AMD

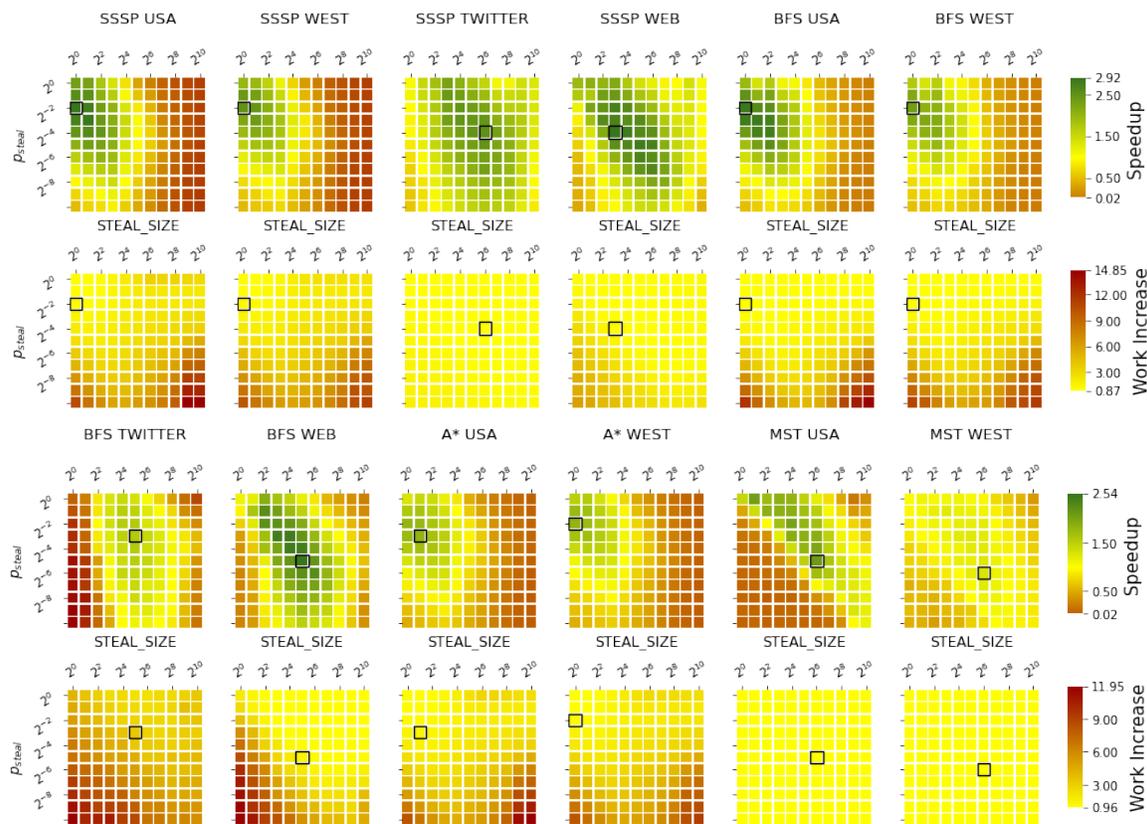


Figure 17: Ablation of stealing probability  $p_{steal}$  and steal buffer size on the **AMD** architecture, for **SMQ** implemented using  $d$ -ary heaps. Experiments execute on 256 threads. The baseline is the classic Multi-Queue on 256 threads with  $C = 4$ . The fastest configuration for each benchmark is highlighted with a black border and listed in Table 12. Best viewed in color.

	<b>SSSP USA</b>	<b>SSSP W</b>	<b>SSSP TWITTER</b>	<b>SSSP WEB</b>	<b>BFS USA</b>	<b>BFS W</b>
$p_{steal}$	1/4	1/4	1/16	1/16	1/4	1/4
CHUNK_SIZE	1	1	64	8	1	1
Speedup	2.28	2.00	2.01	2.92	2.87	2.33
Work Increase	1.18	1.20	1.03	1.11	1.05	1.12
	<b>BFS TWITTER</b>	<b>BFS WEB</b>	<b>A* USA</b>	<b>A* W</b>	<b>MST USA</b>	<b>MST W</b>
$p_{steal}$	1/8	1/32	1/8	1/4	1/32	1/64
CHUNK_SIZE	32	32	2	1	64	64
Speedup	1.30	2.54	1.83	1.82	2.13	1.28
Work Increase	1.33	1.09	1.35	1.23	1.00	1.00

Table 12: The optimal parameters for Stealing Multi-Queue via  $d$ -ary heaps on 256 threads, obtained on the **AMD** platform. Based on Figure 17. For each benchmark, the best parameters are presented with the speedup and work increase.

## D.2 SMQ via D-Ary Heaps on Intel

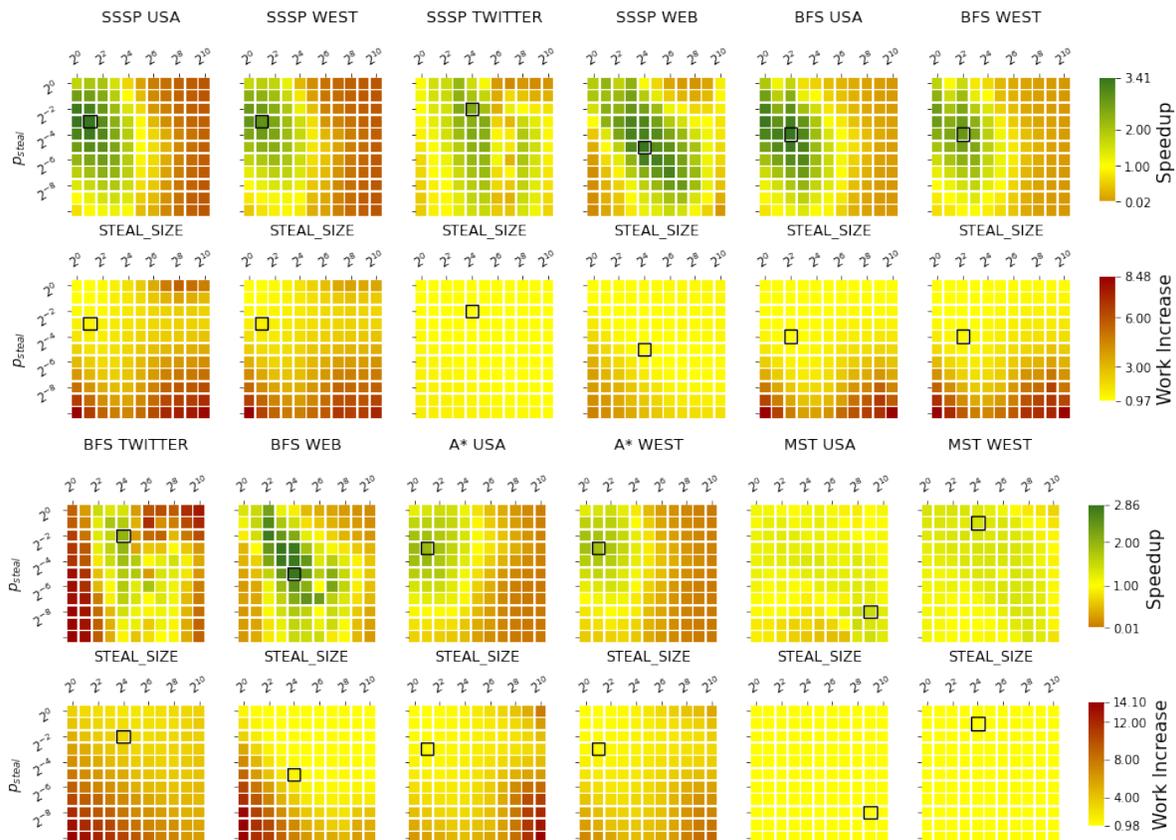


Figure 18: Ablation of stealing probability  $p_{steal}$  and steal buffer size on **Intel**, for *SMQ* implemented using  $d$ -ary heaps. Experiments execute on 128 threads. The baseline is the classic Multi-Queue on 128 threads with  $C = 4$ . The fastest configuration for each benchmark is highlighted with a black border and listed in Table 13. Best viewed in color.

	<b>SSSP USA</b>	<b>SSSP W</b>	<b>SSSP TWITTER</b>	<b>SSSP WEB</b>	<b>BFS USA</b>	<b>BFS W</b>
$p_{steal}$	1/8	1/8	1/4	1/32	1/16	1/16
CHUNK_SIZE	2	2	16	16	4	4
Speedup	2.47	2.15	1.92	2.46	3.41	2.82
Work Increase	1.23	1.38	1.01	1.06	1.06	1.17
	<b>BFS TWITTER</b>	<b>BFS WEB</b>	<b>A* USA</b>	<b>A* W</b>	<b>MST USA</b>	<b>MST W</b>
$p_{steal}$	1/4	1/32	1/8	1/8	1/256	1/2
CHUNK_SIZE	16	16	2	2	512	16
Speedup	1.44	2.86	1.95	1.85	1.48	1.39
Work Increase	1.20	1.10	1.24	1.33	1.00	1.00

Table 13: The optimal parameters of *SMQ* via  $d$ -ary heaps on **Intel**, based on Figure 18. For each benchmark, the best  $p_{steal}$  and steal buffer size combination is presented, with its speedup and work increase.

### D.3 SMQ via Skip Lists on AMD

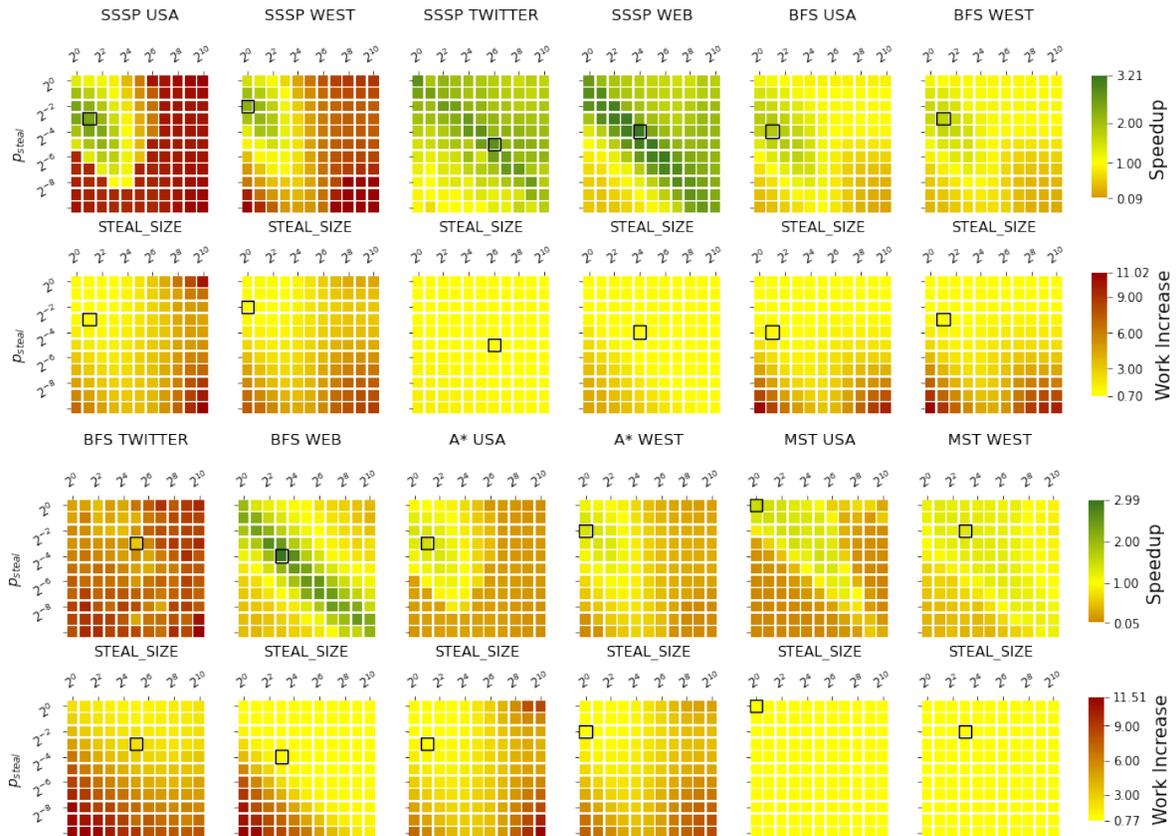


Figure 19: Ablation of stealing probability  $p_{steal}$  and steal buffer size on the AMD architecture, for SMQ implemented using skip-lists. Experiments execute on 256 threads. The baseline is the classic Multi-Queue on 256 threads with  $C = 4$ . The fastest configuration for each benchmark is highlighted with a black border and listed in Table 14. Best viewed in color.

	<b>SSSP USA</b>	<b>SSSP W</b>	<b>SSSP TWITTER</b>	<b>SSSP WEB</b>	<b>BFS USA</b>	<b>BFS W</b>
$p_{steal}$	1/8	1/4	1/32	1/16	1/16	1/8
CHUNK_SIZE	2	1	64	16	2	2
Speedup	1.59	1.51	1.74	3.21	1.87	1.68
Work Increase	1.13	1.00	1.03	1.01	1.21	1.13
	<b>BFS TWITTER</b>	<b>BFS WEB</b>	<b>A* USA</b>	<b>A* W</b>	<b>MST USA</b>	<b>MST W</b>
$p_{steal}$	1/8	1/16	1/8	1/4	1/1	1/4
CHUNK_SIZE	32	8	2	1	1	8
Speedup	0.88	2.99	1.44	1.42	1.52	1.34
Work Increase	1.14	1.04	1.14	1.04	1.01	1.00

Table 14: The optimal parameters of SMQ via skip lists on the AMD architecture, based on Figure 17. For each benchmark, the best  $p_{steal}$  and steal buffer size combination is presented, with its speedup and work increase.

## D.4 SMQ via Skip Lists on Intel

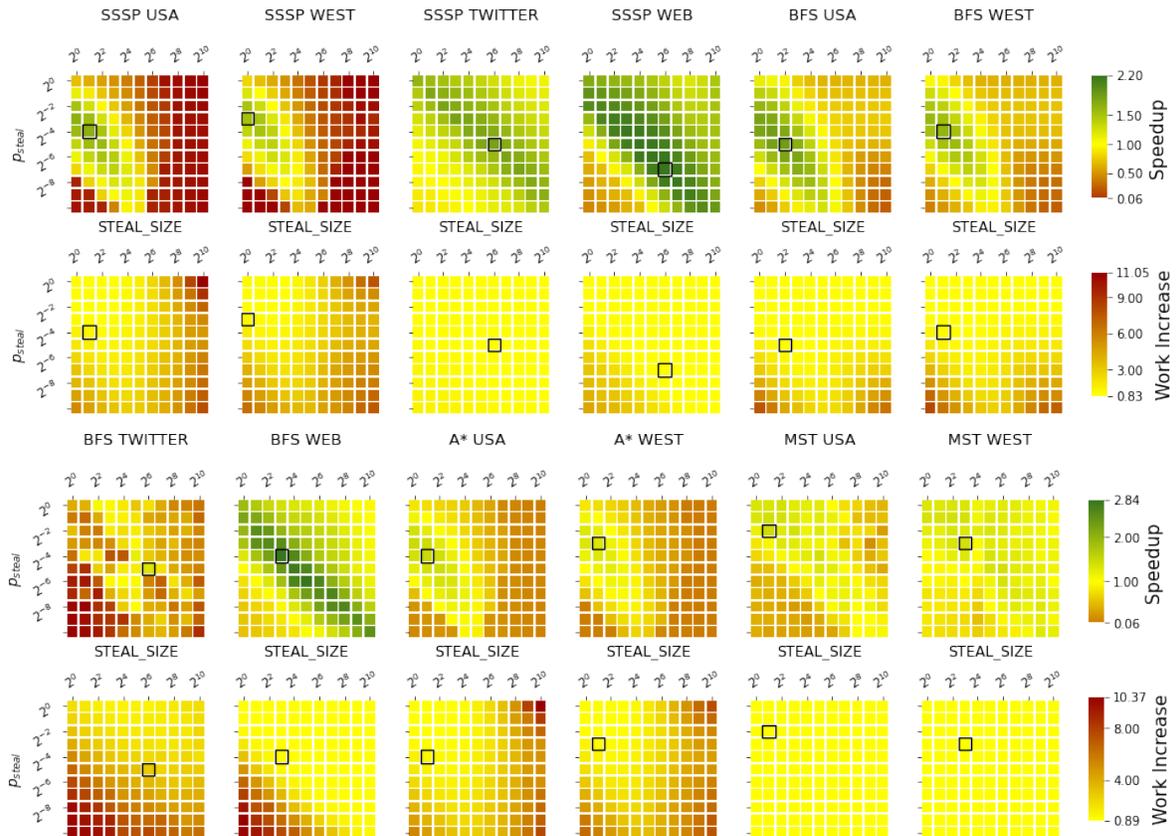


Figure 20: Ablation of stealing probability  $p_{steal}$  and steal buffer size on **Intel**, for *SMQ* implemented using skip-lists. Experiments execute on 128 threads. The baseline is the classic Multi-Queue on 128 threads with  $C = 4$ . The fastest configuration for each benchmark is highlighted with a black border and listed in Table 15. Best viewed in color.

	<b>SSSP USA</b>	<b>SSSP W</b>	<b>SSSP TWITTER</b>	<b>SSSP WEB</b>	<b>BFS USA</b>	<b>BFS W</b>
$p_{steal}$	1/16	1/8	1/32	1/128	1/32	1/16
CHUNK_SIZE	2	1	64	64	4	2
Speedup	1.53	1.42	1.64	2.20	1.85	1.64
Work Increase	1.28	1.30	1.01	1.02	1.12	1.18
	<b>BFS TWITTER</b>	<b>BFS WEB</b>	<b>A* USA</b>	<b>A* W</b>	<b>MST USA</b>	<b>MST W</b>
$p_{steal}$	1/32	1/16	1/16	1/8	1/4	1/8
CHUNK_SIZE	64	8	2	2	2	8
Speedup	1.10	2.84	1.49	1.33	1.43	1.42
Work Increase	1.21	1.02	1.25	1.14	1.01	1.00

Table 15: The optimal parameters of *SMQ* via skip lists on the **Intel** architecture, based on Figure 20. For each benchmark, the best  $p_{steal}$  and steal buffer size combination is presented, with its speedup and work increase.

## E NUMA Awareness

In this section we examine how the proposed in Section 4 optimization for NUMA architectures improves both the Multi-Queue variants (Tables 16–23) and the proposed Stealing Multi-Queue (SMQ) via both d-ary heaps (Tables 24–25) and Skip-Lists (Tables 26–27).

### E.1 MQ Optimized NUMA: insert=Temporal Locality, delete=Temporal Locality

	1	2	4	8	16	32	64	128	256	512	1024
<b>BFS USA</b>	1.44	1.56	1.65	1.70	1.73	<b>1.75</b>	<b>1.75</b>	1.73	1.64	1.54	1.36
<b>BFS WEST</b>	1.21	1.30	1.37	1.41	1.40	<b>1.48</b>	<b>1.48</b>	1.47	1.41	1.32	1.20
<b>BFS TWITTER</b>	<b>1.14</b>	1.10	1.12	1.11	1.04	0.97	0.91	0.88	0.82	0.80	0.78
<b>BFS WEB</b>	2.22	2.32	<b>2.38</b>	2.03	1.97	1.67	1.52	1.55	1.43	1.44	1.45
<b>SSSP USA</b>	1.00	1.03	1.10	1.12	1.16	1.19	1.19	<b>1.20</b>	<b>1.20</b>	<b>1.20</b>	1.15
<b>SSSP WEST</b>	0.95	0.98	0.99	1.05	1.08	1.09	1.12	<b>1.14</b>	1.09	1.06	1.08
<b>SSSP TWITTER</b>	1.30	1.27	<b>1.35</b>	1.28	1.23	1.27	1.27	1.27	1.33	1.23	1.33
<b>SSSP WEB</b>	1.68	1.77	1.78	<b>1.86</b>	1.78	1.67	1.60	1.55	1.44	1.55	1.58
<b>MST USA</b>	0.43	1.06	<b>1.15</b>	1.00	0.98	0.79	0.59	0.69	0.40	0.47	0.73
<b>MST WEST</b>	<b>1.16</b>	1.12	1.06	1.12	1.12	1.13	1.13	1.14	1.14	1.15	1.09
<b>A* USA</b>	0.95	1.00	1.05	1.09	1.12	1.12	1.13	<b>1.15</b>	1.13	1.11	1.11
<b>A* WEST</b>	0.94	1.02	1.06	1.08	1.11	1.13	<b>1.17</b>	1.16	<b>1.17</b>	1.14	1.11

Table 16: Speedups for Multi-Queue variants with the *temporal locality* optimization for both `insert(..)` and `delete()`, and various weights  $K$  for non-local NUMA node accesses, obtained on the **AMD** platform on 256 threads; the best speedups are highlighted with **red**. The baseline is the classic Multi-Queue on 256 threads with  $C = 4$ . With  $K = 1$  the algorithm is the same as without the NUMA-specific optimization.

	1	2	4	8	16	32	64	128	256	512	1024
<b>BFS USA</b>	1.58	1.79	2.09	2.37	2.64	2.84	<b>2.97</b>	2.86	2.72	2.49	1.99
<b>BFS WEST</b>	1.28	1.45	1.68	1.94	2.16	2.37	<b>2.43</b>	2.40	2.17	2.02	1.68
<b>BFS TWITTER</b>	<b>1.58</b>	<b>1.58</b>	1.39	1.36	1.21	1.05	0.99	0.78	0.73	0.64	0.68
<b>BFS WEB</b>	3.11	3.32	<b>3.69</b>	3.23	2.62	2.14	1.81	1.56	1.53	1.31	1.26
<b>SSSP USA</b>	0.92	0.99	1.12	1.28	1.44	1.58	1.68	1.74	<b>1.78</b>	1.77	1.72
<b>SSSP WEST</b>	0.88	0.92	1.02	1.17	1.32	1.46	1.56	1.61	<b>1.65</b>	1.62	1.54
<b>SSSP TWITTER</b>	1.77	1.90	<b>1.96</b>	<b>1.96</b>	1.63	1.74	1.74	1.72	1.74	1.67	1.66
<b>SSSP WEB</b>	2.11	2.30	<b>2.34</b>	2.25	1.90	1.62	1.38	1.18	1.09	1.00	0.95
<b>MST USA</b>	1.02	1.11	1.22	<b>1.28</b>	<b>1.28</b>	<b>1.28</b>	1.22	1.20	1.22	1.22	1.18
<b>MST WEST</b>	1.16	1.17	1.20	1.10	1.12	1.24	<b>1.31</b>	1.28	1.28	1.21	1.13
<b>A* USA</b>	0.99	1.06	1.19	1.34	1.49	1.60	1.69	1.75	<b>1.76</b>	1.75	1.69
<b>A* WEST</b>	0.93	0.97	1.07	1.22	1.36	1.48	1.58	1.63	<b>1.65</b>	1.64	1.58

Table 17: Speedups for Multi-Queue variants with the *temporal locality* optimization for both `insert(..)` and `delete()`, and various weights  $K$  for non-local NUMA node accesses, obtained on the **Intel** platform on 128 threads; the best speedups are highlighted with **red**. The baseline is the classic Multi-Queue on 128 threads with  $C = 4$ . With  $K = 1$  the algorithm is the same as without the NUMA-specific optimization.

## E.2 MQ Optimized NUMA: insert=Temporal Locality, delete=Task Batching

	1	2	4	8	16	32	64	128	256	512	1024
<b>BFS USA</b>	1.99	2.06	2.04	2.14	2.10	2.16	2.15	2.16	<b>2.18</b>	2.10	2.03
<b>BFS WEST</b>	1.56	1.64	1.67	1.71	1.70	1.72	1.73	<b>1.76</b>	1.72	1.68	1.60
<b>BFS TWITTER</b>	1.15	1.15	<b>1.17</b>	1.16	1.13	1.11	1.10	1.07	1.05	1.00	0.95
<b>BFS WEB</b>	2.21	2.23	2.27	2.33	<b>2.38</b>	2.37	2.11	2.01	1.77	1.78	1.62
<b>SSSP USA</b>	1.05	1.13	1.18	1.20	1.26	1.28	1.29	1.29	<b>1.31</b>	1.30	1.27
<b>SSSP WEST</b>	0.88	0.95	1.00	1.04	1.05	1.07	1.07	1.08	<b>1.10</b>	1.09	1.07
<b>SSSP TWITTER</b>	1.16	1.26	1.22	1.32	1.29	1.29	1.25	<b>1.33</b>	1.28	<b>1.33</b>	1.26
<b>SSSP WEB</b>	1.58	1.56	1.68	1.71	1.72	<b>1.81</b>	1.70	1.55	1.62	1.62	1.54
<b>MST USA</b>	1.13	1.15	<b>1.20</b>	1.12	1.18	1.17	1.12	0.89	0.69	0.61	0.43
<b>MST WEST</b>	<b>1.17</b>	1.14	1.13	1.13	1.13	1.13	1.12	1.14	1.13	1.14	1.13
<b>A* USA</b>	0.97	1.07	1.10	1.15	1.18	1.17	1.21	<b>1.22</b>	<b>1.22</b>	1.19	1.20
<b>A* WEST</b>	0.98	0.97	1.06	1.05	1.13	1.16	1.16	<b>1.19</b>	1.17	1.11	1.12

Table 18: Speedups for Multi-Queue variants with the *temporal locality* optimization for `insert(·)`, *task batching* for `delete()`, and various weights  $K$  for non-local NUMA node accesses, obtained on the **AMD** platform on 256 threads; the best speedups are highlighted with **red**. The baseline is the classic Multi-Queue on 256 threads with  $C = 4$ . With  $K = 1$  the algorithm is the same as without the NUMA-specific optimization.

	1	2	4	8	16	32	64	128	256	512	1024
<b>BFS USA</b>	2.22	2.37	2.58	2.75	2.93	3.08	3.22	<b>3.37</b>	3.35	3.28	3.15
<b>BFS WEST</b>	1.79	1.93	2.11	2.32	2.47	2.64	2.77	2.84	<b>2.86</b>	2.78	2.67
<b>BFS TWITTER</b>	<b>1.69</b>	1.55	<b>1.69</b>	1.62	1.55	1.54	1.50	1.42	1.35	1.32	1.28
<b>BFS WEB</b>	3.47	3.60	3.65	<b>3.68</b>	3.61	3.44	3.62	3.13	2.64	2.69	2.66
<b>SSSP USA</b>	1.07	1.14	1.25	1.39	1.52	1.66	1.76	1.84	1.89	<b>1.90</b>	1.89
<b>SSSP WEST</b>	0.95	1.01	1.11	1.23	1.36	1.47	1.55	1.62	<b>1.66</b>	<b>1.66</b>	1.63
<b>SSSP TWITTER</b>	1.56	1.61	1.80	1.82	1.93	1.87	<b>1.96</b>	1.83	1.74	1.69	1.71
<b>SSSP WEB</b>	2.19	2.21	2.32	2.30	2.45	<b>2.53</b>	2.48	2.10	1.91	1.78	1.67
<b>MST USA</b>	1.15	1.21	1.22	1.25	1.23	1.27	<b>1.30</b>	1.23	1.20	1.08	1.09
<b>MST WEST</b>	1.20	1.19	1.26	1.27	1.24	1.30	1.28	<b>1.31</b>	1.30	1.10	1.27
<b>A* USA</b>	1.18	1.24	1.32	1.43	1.55	1.62	1.70	1.74	<b>1.79</b>	<b>1.79</b>	1.78
<b>A* WEST</b>	1.02	1.07	1.16	1.28	1.41	1.50	1.59	1.64	<b>1.68</b>	1.67	1.65

Table 19: Speedups for Multi-Queue variants with the *temporal locality* optimization for `insert(·)`, *task batching* for `delete()`, and various weights  $K$  for non-local NUMA node accesses, obtained on the **Intel** platform on 128 threads; the best speedups are highlighted with **red**. The baseline is the classic Multi-Queue on 128 threads with  $C = 4$ . With  $K = 1$  the algorithm is the same as without the NUMA-specific optimization.

### E.3 MQ Optimized NUMA: insert=Task Batching, delete=Temporal Locality

	1	2	4	8	16	32	64	128	256	512	1024
<b>BFS USA</b>	1.52	1.58	1.70	1.78	1.74	1.84	1.85	<b>1.86</b>	1.81	1.81	1.77
<b>BFS WEST</b>	1.23	1.28	1.33	1.43	1.40	1.46	<b>1.51</b>	1.49	1.47	1.47	1.39
<b>BFS TWITTER</b>	1.13	<b>1.18</b>	<b>1.18</b>	1.11	1.11	1.06	1.00	0.93	0.86	0.83	0.80
<b>BFS WEB</b>	2.03	2.09	2.25	<b>2.36</b>	2.08	1.85	1.76	1.57	1.53	1.43	1.44
<b>SSSP USA</b>	0.99	<b>1.00</b>	0.97	0.90	0.95	0.95	0.97	<b>1.00</b>	0.85	0.94	0.97
<b>SSSP WEST</b>	0.81	0.82	0.79	0.74	0.81	<b>0.85</b>	0.81	0.80	0.84	0.81	0.78
<b>SSSP TWITTER</b>	1.18	1.27	1.33	1.31	1.26	1.25	1.30	1.28	1.25	1.26	<b>1.37</b>
<b>SSSP WEB</b>	1.53	1.59	1.72	1.80	<b>1.87</b>	1.85	1.79	1.50	1.52	1.45	1.45
<b>MST USA</b>	1.16	1.14	1.14	1.05	<b>1.18</b>	1.09	0.95	0.98	0.88	0.89	0.91
<b>MST WEST</b>	1.07	1.06	1.08	1.09	1.11	1.13	1.14	1.18	1.14	<b>1.19</b>	1.18
<b>A* USA</b>	0.89	0.88	0.90	0.88	0.89	0.92	0.91	<b>0.94</b>	0.91	0.91	0.90
<b>A* WEST</b>	0.97	0.99	1.04	1.08	1.10	1.06	<b>1.15</b>	1.11	1.14	1.13	1.11

Table 20: Speedups for Multi-Queue variants with the *task batching* optimization for `insert(...)`, *temporal locality* for `delete()`, and various weights  $K$  for non-local NUMA node accesses, obtained on the **AMD** platform on 256 threads; the best speedups are highlighted with **red**. The baseline is the classic Multi-Queue on 256 threads with  $C = 4$ . With  $K = 1$  the algorithm is the same as without the NUMA-specific optimization.

	1	2	4	8	16	32	64	128	256	512	1024
<b>BFS USA</b>	1.68	1.81	2.01	2.26	2.48	2.66	2.82	2.92	<b>2.95</b>	2.87	2.73
<b>BFS WEST</b>	1.32	1.45	1.63	1.87	2.08	2.26	2.38	<b>2.47</b>	2.42	2.36	2.09
<b>BFS TWITTER</b>	1.54	1.52	<b>1.56</b>	1.45	1.30	1.20	1.10	0.96	0.87	0.74	0.67
<b>BFS WEB</b>	2.93	3.07	3.30	<b>3.46</b>	3.43	2.81	2.38	1.91	1.68	1.42	1.25
<b>SSSP USA</b>	0.68	0.74	0.84	0.98	1.09	1.19	1.25	1.29	<b>1.30</b>	1.27	1.24
<b>SSSP WEST</b>	0.65	0.66	0.78	0.92	1.03	1.12	1.19	1.22	<b>1.23</b>	1.21	1.15
<b>SSSP TWITTER</b>	1.82	1.98	2.06	<b>2.14</b>	2.04	2.00	1.81	1.80	1.83	1.77	1.83
<b>SSSP WEB</b>	2.04	2.18	2.38	2.46	<b>2.61</b>	2.39	2.10	1.70	1.44	1.19	1.07
<b>MST USA</b>	<b>1.18</b>	1.15	1.09	1.17	1.17	1.07	0.81	0.60	0.34	0.11	0.34
<b>MST WEST</b>	0.90	0.98	0.91	0.89	1.06	1.03	0.93	0.95	<b>1.15</b>	1.01	0.92
<b>A* USA</b>	0.77	0.82	0.93	1.05	1.17	1.25	1.32	<b>1.36</b>	<b>1.36</b>	1.34	1.29
<b>A* WEST</b>	0.71	0.75	0.85	0.97	1.08	1.16	1.22	<b>1.26</b>	1.25	1.23	1.15

Table 21: Speedups for Multi-Queue variants with the *task batching* optimization for `insert(...)`, *temporal locality* for `delete()`, and various weights  $K$  for non-local NUMA node accesses, obtained on the **Intel** platform on 128 threads; the best speedups are highlighted with **red**. The baseline is the classic Multi-Queue on 128 threads with  $C = 4$ . With  $K = 1$  the algorithm is the same as without the NUMA-specific optimization.

#### E.4 MQ Optimized NUMA: insert=Task Batching, delete=Task Batching

	1	2	4	8	16	32	64	128	256	512	1024
<b>BFS USA</b>	2.09	2.10	2.15	2.18	2.23	2.26	<b>2.27</b>	2.22	2.25	2.22	2.10
<b>BFS WEST</b>	1.62	1.64	1.66	1.40	1.69	1.70	1.71	1.65	<b>1.75</b>	1.71	1.57
<b>BFS TWITTER</b>	1.15	1.17	<b>1.18</b>	1.16	1.14	1.12	1.06	1.06	1.07	1.05	1.00
<b>BFS WEB</b>	2.27	2.23	2.37	2.44	2.49	<b>2.52</b>	<b>2.52</b>	2.30	2.16	1.98	1.78
<b>SSSP USA</b>	1.10	1.13	1.16	1.20	1.22	1.22	1.25	1.24	1.22	<b>1.26</b>	1.24
<b>SSSP WEST</b>	0.96	0.98	1.02	1.09	1.11	1.16	<b>1.18</b>	1.14	1.12	1.13	1.16
<b>SSSP TWITTER</b>	1.18	1.17	1.27	1.29	1.32	1.21	1.29	1.32	1.27	1.23	<b>1.33</b>
<b>SSSP WEB</b>	1.42	1.43	1.48	1.53	1.59	<b>1.64</b>	1.61	1.59	1.49	1.41	1.39
<b>MST USA</b>	1.15	<b>1.17</b>	1.12	1.16	1.11	1.12	1.14	1.11	1.11	1.07	1.06
<b>MST WEST</b>	1.15	1.15	1.15	1.16	1.15	1.18	1.18	1.18	1.14	<b>1.19</b>	<b>1.19</b>
<b>A* USA</b>	1.05	1.09	1.10	1.13	1.15	1.19	1.20	1.20	1.23	1.19	<b>1.24</b>
<b>A* WEST</b>	0.98	1.03	1.08	1.10	1.14	1.19	1.20	<b>1.21</b>	<b>1.21</b>	<b>1.21</b>	1.18

Table 22: Speedups for Multi-Queue variants with the *task batching* optimization for both `insert(...)` and `delete()`, and various weights  $K$  for non-local NUMA node accesses, obtained on the **AMD** platform on 256 threads; the best speedups are highlighted with **red**. The baseline is the classic Multi-Queue on 256 threads with  $C = 4$ . With  $K = 1$  the algorithm is the same as without the NUMA-specific optimization.

	1	2	4	8	16	32	64	128	256	512	1024
<b>BFS USA</b>	2.47	2.50	2.63	2.79	2.99	3.16	3.33	<b>3.44</b>	3.42	3.35	3.17
<b>BFS WEST</b>	1.87	1.94	2.04	2.17	2.38	2.51	2.64	<b>2.73</b>	2.56	2.72	2.58
<b>BFS TWITTER</b>	<b>1.64</b>	1.59	1.58	1.62	1.61	1.57	1.50	1.51	1.49	1.41	1.43
<b>BFS WEB</b>	3.49	3.44	3.47	3.61	3.41	<b>3.72</b>	3.44	3.45	3.55	3.33	3.33
<b>SSSP USA</b>	1.13	1.16	1.23	1.33	1.44	1.54	1.64	1.72	1.79	1.85	<b>1.88</b>
<b>SSSP WEST</b>	0.88	0.88	0.98	1.05	1.22	1.26	1.34	1.37	1.32	<b>1.38</b>	1.35
<b>SSSP TWITTER</b>	1.69	1.68	1.67	1.80	1.81	<b>1.91</b>	1.88	1.81	1.72	1.69	1.65
<b>SSSP WEB</b>	2.08	2.10	2.14	2.22	2.34	2.36	<b>2.42</b>	2.26	1.91	1.84	1.62
<b>MST USA</b>	1.00	1.05	1.07	1.13	1.09	<b>1.15</b>	1.10	1.09	1.07	1.00	0.78
<b>MST WEST</b>	1.10	1.03	0.95	1.03	1.13	1.08	<b>1.19</b>	1.13	1.04	1.13	1.09
<b>A* USA</b>	1.19	1.21	1.28	1.37	1.47	1.56	1.64	1.71	1.79	1.82	<b>1.86</b>
<b>A* WEST</b>	0.97	1.00	1.09	1.21	1.23	1.31	1.41	1.45	1.45	<b>1.55</b>	1.52

Table 23: Speedups for Multi-Queue variants with the *task batching* optimization for both `insert(...)` and `delete()`, and various weights  $K$  for non-local NUMA node accesses, obtained on the **Intel** platform on 128 threads; the best speedups are highlighted with **red**. The baseline is the classic Multi-Queue on 128 threads with  $C = 4$ . With  $K = 1$  the algorithm is the same as without the NUMA-specific optimization.

## E.5 SMQ via D-Ary Heaps NUMA

	1	2	4	8	16	32	64	128	256	512	1024
<b>BFS USA</b>	2.85	2.87	2.88	<b>2.89</b>	2.88	2.78	2.86	2.87	2.88	2.86	2.88
<b>BFS WEST</b>	2.35	2.38	2.39	<b>2.42</b>	2.37	2.33	2.35	2.38	2.39	2.37	2.41
<b>BFS TWITTER</b>	1.28	1.28	1.29	1.28	1.28	<b>1.30</b>	<b>1.30</b>	<b>1.30</b>	<b>1.30</b>	<b>1.30</b>	1.28
<b>BFS WEB</b>	2.55	2.36	2.59	2.47	2.62	2.56	<b>2.63</b>	<b>2.63</b>	2.54	2.60	2.60
<b>SSSP USA</b>	2.35	<b>2.44</b>	2.43	<b>2.44</b>	<b>2.44</b>	2.41	2.43	2.43	2.40	2.43	2.43
<b>SSSP WEST</b>	2.12	2.10	2.10	2.11	2.12	2.10	2.11	2.10	2.11	<b>2.13</b>	2.11
<b>SSSP TWITTER</b>	2.04	2.01	2.05	1.94	2.02	2.01	1.99	2.05	2.00	<b>2.06</b>	2.02
<b>SSSP WEB</b>	2.63	2.77	2.67	2.74	2.60	2.76	2.86	2.62	2.80	2.78	<b>2.88</b>
<b>MST USA</b>	1.99	1.90	1.94	1.82	1.85	1.96	1.89	1.83	1.95	<b>2.09</b>	1.74
<b>MST WEST</b>	1.26	1.22	1.28	1.27	1.27	1.27	1.30	1.29	1.27	<b>1.31</b>	1.25
<b>A* USA</b>	<b>1.92</b>	1.89	1.90	1.87	1.90	1.89	1.90	1.90	1.91	1.91	1.91
<b>A* WEST</b>	1.92	<b>1.94</b>	1.93	1.87	1.92	1.91	1.90	1.84	1.91	1.92	1.90

Table 24: Speedups for the Stealing Multi-Queue implementation via  $d$ -ary heaps with various weights  $K$  for non-local NUMA node accesses, obtained on the **AMD** platform on 256 threads; the best speedups are highlighted with **red**. The baseline is the classic Multi-Queue on 256 threads with  $C = 4$ . With  $K = 1$  the algorithm is the same as without the NUMA-specific optimization.

	1	2	4	8	16	32	64	128	256	512	1024
<b>BFS USA</b>	3.27	<b>3.28</b>	3.27	3.19	3.27	<b>3.28</b>	3.27	3.26	3.27	3.27	3.27
<b>BFS WEST</b>	<b>2.72</b>	<b>2.72</b>	2.71	2.71	<b>2.72</b>	2.71	2.71	2.70	2.71	2.71	2.71
<b>BFS TWITTER</b>	1.50	1.48	1.46	1.50	<b>1.51</b>	<b>1.51</b>	1.46	1.46	1.49	1.47	1.49
<b>BFS WEB</b>	2.85	2.78	3.04	2.99	2.90	2.67	2.97	2.68	3.00	2.94	<b>3.08</b>
<b>SSSP USA</b>	<b>2.35</b>	<b>2.35</b>	2.34	<b>2.35</b>	<b>2.35</b>	<b>2.35</b>	2.34	<b>2.35</b>	<b>2.35</b>	<b>2.35</b>	2.34
<b>SSSP WEST</b>	2.02	<b>2.03</b>	<b>2.03</b>	<b>2.03</b>	2.01	2.01	2.01	<b>2.03</b>	2.01	2.02	2.01
<b>SSSP TWITTER</b>	1.92	1.86	1.92	<b>1.93</b>	1.72	1.89	1.86	1.90	1.78	1.91	1.83
<b>SSSP WEB</b>	2.43	2.37	2.33	2.46	<b>2.47</b>	2.40	2.29	2.44	2.44	2.42	2.30
<b>MST USA</b>	1.24	1.27	1.20	1.25	<b>1.31</b>	1.11	1.28	1.25	1.28	1.18	1.22
<b>MST WEST</b>	1.21	1.21	1.25	1.27	1.21	1.22	1.19	1.26	<b>1.28</b>	1.27	1.24
<b>A* USA</b>	2.14	2.15	2.16	<b>2.17</b>	2.15	2.16	2.16	2.16	2.15	2.16	2.16
<b>A* WEST</b>	1.90	1.91	1.91	1.91	1.91	<b>1.92</b>	1.90	1.90	1.90	<b>1.92</b>	<b>1.92</b>

Table 25: Speedups for the Stealing Multi-Queue implementation via  $d$ -ary heaps with various weights  $K$  for non-local NUMA node accesses, obtained on the **Intel** platform on 128 threads; the best speedups are highlighted with **red**. The baseline is the classic Multi-Queue on 128 threads with  $C = 4$ . With  $K = 1$  the algorithm is the same as without the NUMA-specific optimization.

## E.6 SMQ via Skip Lists NUMA

	1	2	4	8	16	32	64	128	256	512	1024
<b>BFS USA</b>	1.80	1.72	1.81	1.82	<b>1.83</b>	1.80	1.82	1.81	1.81	1.82	1.80
<b>BFS WEST</b>	1.55	1.60	1.42	1.61	1.60	1.60	1.58	1.60	<b>1.62</b>	1.52	1.60
<b>BFS TWITTER</b>	0.94	0.91	0.94	0.83	0.93	0.95	0.86	<b>0.97</b>	0.87	0.75	<b>0.97</b>
<b>BFS WEB</b>	<b>3.24</b>	3.17	2.66	3.08	2.81	3.15	2.96	3.06	2.48	3.12	2.72
<b>SSSP USA</b>	1.68	<b>1.73</b>	1.68	1.72	1.66	1.66	1.71	1.72	1.71	1.70	1.71
<b>SSSP WEST</b>	1.52	1.54	1.53	1.55	1.55	1.53	1.54	<b>1.56</b>	1.55	<b>1.56</b>	1.55
<b>SSSP TWITTER</b>	1.87	1.89	<b>1.99</b>	1.70	1.77	1.69	1.85	1.64	1.78	1.63	1.78
<b>SSSP WEB</b>	3.28	3.25	3.13	3.18	<b>3.29</b>	3.23	3.20	3.19	3.15	3.20	3.20
<b>MST USA</b>	1.42	1.40	1.45	<b>1.46</b>	1.44	<b>1.46</b>	1.44	1.29	1.45	1.44	1.45
<b>MST WEST</b>	1.22	<b>1.32</b>	<b>1.32</b>	1.31	1.31	1.27	1.20	1.25	1.25	1.18	1.23
<b>A* USA</b>	1.49	1.50	1.50	<b>1.51</b>	1.50	1.50	1.50	1.50	1.49	1.50	1.50
<b>A* WEST</b>	<b>1.47</b>	<b>1.47</b>	1.46	<b>1.47</b>	1.45	<b>1.47</b>	1.41	<b>1.47</b>	<b>1.47</b>	<b>1.47</b>	<b>1.47</b>

Table 26: Speedups for the Stealing Multi-Queue implementation via skip lists with various weights  $K$  for non-local NUMA node accesses, obtained on the **AMD** platform on 256 threads; the best speedups are highlighted with **red**. The baseline is the classic Multi-Queue on 256 threads with  $C = 4$ . With  $K = 1$  the algorithm is the same as without the NUMA-specific optimization.

	1	2	4	8	16	32	64	128	256	512	1024
<b>BFS USA</b>	1.83	1.85	1.84	1.84	1.83	1.84	1.84	1.85	<b>1.86</b>	1.85	1.84
<b>BFS WEST</b>	1.62	1.62	1.63	1.63	1.64	<b>1.65</b>	<b>1.65</b>	<b>1.65</b>	<b>1.65</b>	<b>1.65</b>	<b>1.65</b>
<b>BFS TWITTER</b>	0.99	0.95	1.03	<b>1.07</b>	0.97	1.03	1.00	1.02	1.06	0.98	1.06
<b>BFS WEB</b>	<b>2.84</b>	2.35	2.67	<b>2.84</b>	2.82	2.79	2.80	2.68	2.83	2.68	2.65
<b>SSSP USA</b>	1.47	1.47	1.47	<b>1.48</b>	1.47	<b>1.48</b>	<b>1.48</b>	<b>1.48</b>	<b>1.48</b>	<b>1.48</b>	<b>1.48</b>
<b>SSSP WEST</b>	1.35	1.33	1.36	1.36	1.36	1.36	<b>1.37</b>	1.36	1.36	<b>1.37</b>	<b>1.37</b>
<b>SSSP TWITTER</b>	1.66	1.65	1.65	1.64	1.65	<b>1.67</b>	1.65	1.66	1.66	<b>1.67</b>	1.65
<b>SSSP WEB</b>	2.19	2.19	2.21	2.21	2.20	2.19	2.20	2.21	2.19	<b>2.23</b>	2.20
<b>MST USA</b>	1.34	1.33	1.29	1.29	1.30	1.29	1.32	1.36	1.28	<b>1.38</b>	1.28
<b>MST WEST</b>	1.41	1.47	<b>1.48</b>	1.40	1.46	<b>1.48</b>	1.47	1.38	<b>1.48</b>	1.45	<b>1.48</b>
<b>A* USA</b>	1.47	1.47	<b>1.48</b>	1.47	1.47	<b>1.48</b>	1.47	<b>1.48</b>	1.47	1.47	<b>1.48</b>
<b>A* WEST</b>	1.31	<b>1.33</b>	<b>1.33</b>	1.32	<b>1.33</b>						

Table 27: Speedups for the Stealing Multi-Queue implementation via skip lists with various weights  $K$  for non-local NUMA node accesses, obtained on the **Intel** platform on 128 threads; the best speedups are highlighted with **red**. The baseline is the classic Multi-Queue on 128 threads with  $C = 4$ . With  $K = 1$  the algorithm is the same as without the NUMA-specific optimization.

## F Final Results: The Magnified Versions for AMD and Intel

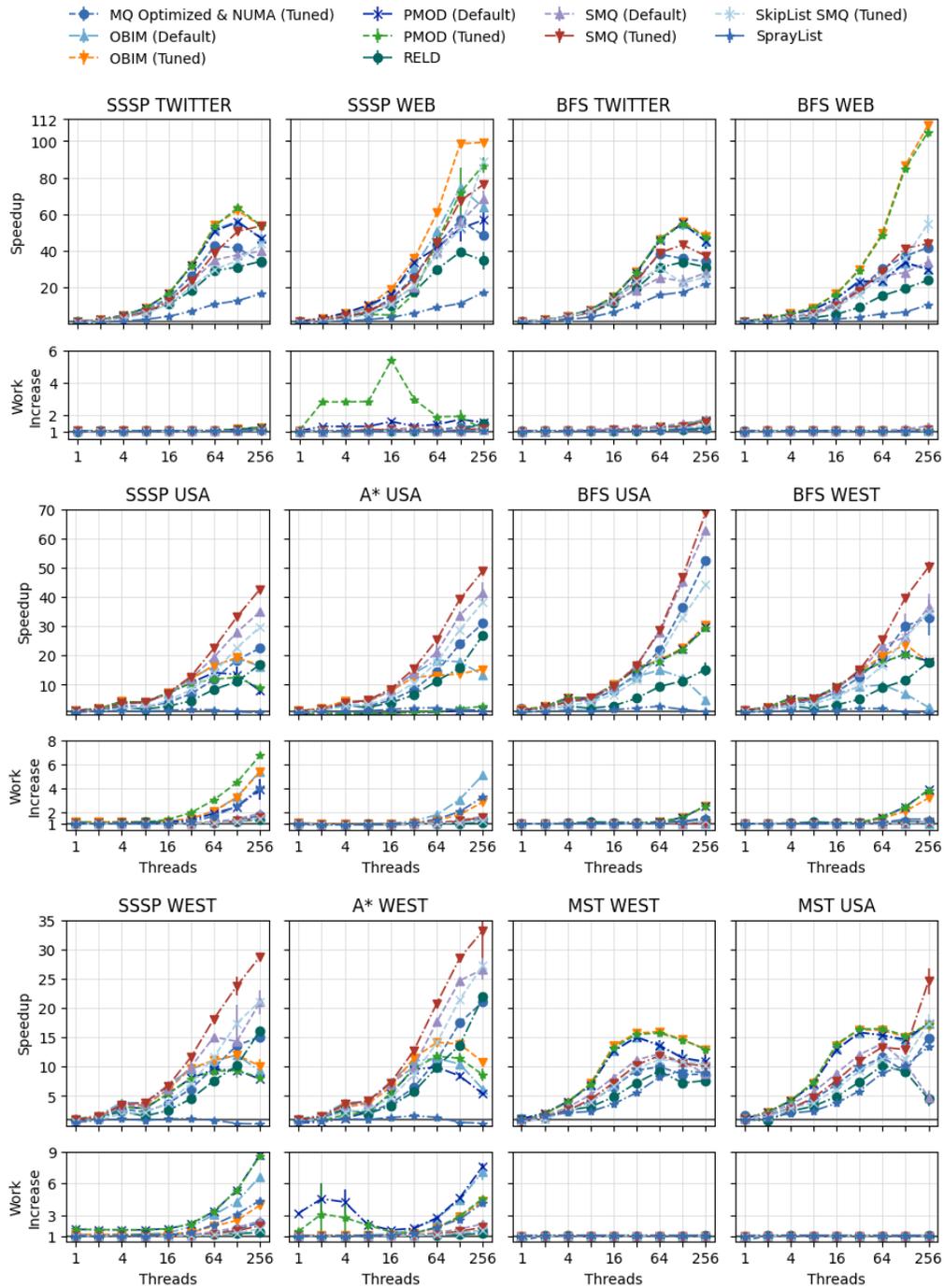


Figure 21: Comparison of the tuned and default variants of SMQ, PMOD, and OBIM, an optimized version of the classic Multi-Queue, SprayList, and RELD schedulers on the **AMD** platform. Speedups are versus the baseline Multi-Queue executed on a single thread. Implementation details are provided in the main body.

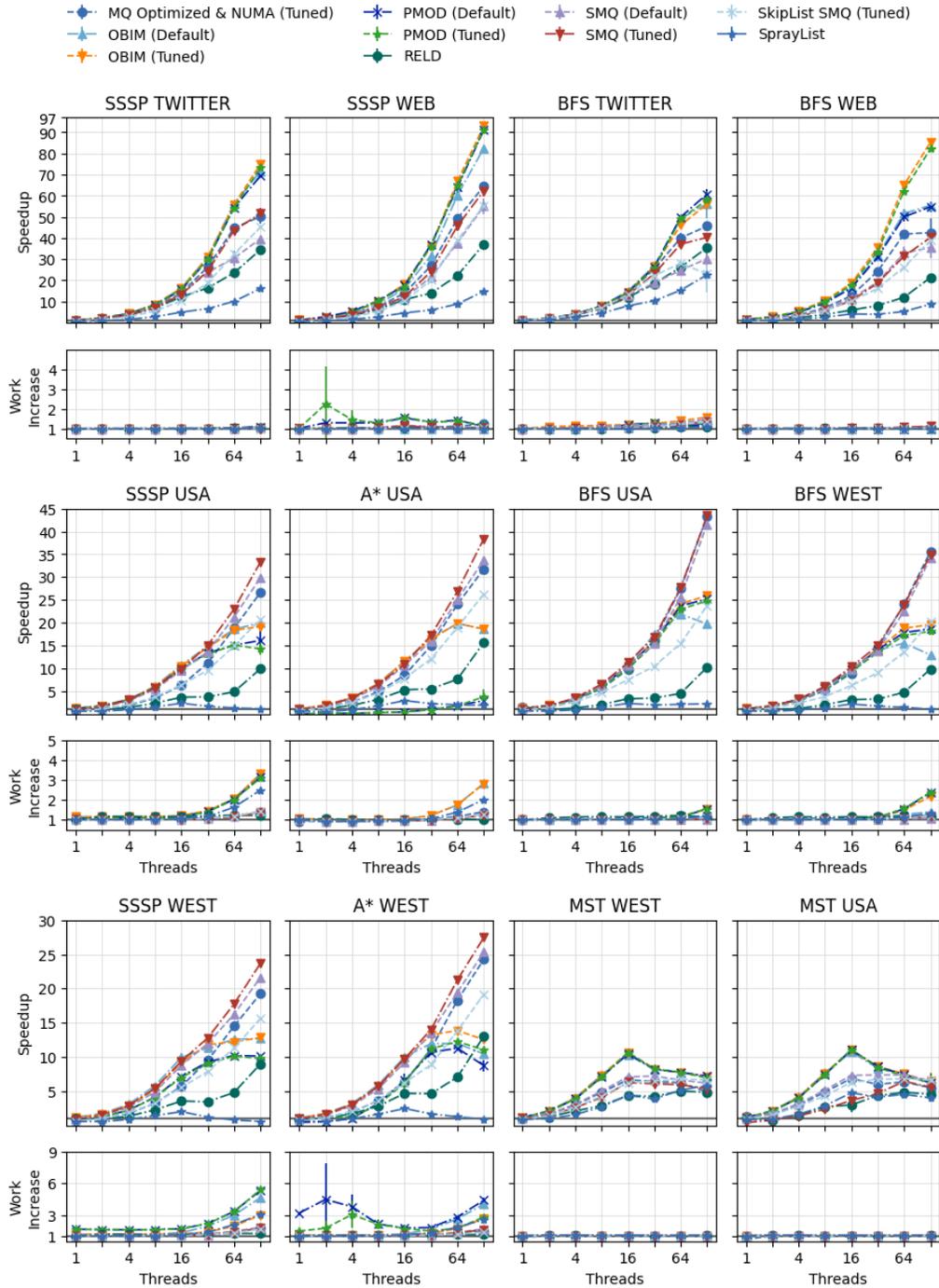


Figure 22: Comparison of the tuned and default variants of SMQ, PMOD, and OBIM, an optimized version of the classic Multi-Queue, SprayList, and RELD schedulers on the **Intel** platform. Speedups are versus the baseline Multi-Queue executed on a single thread. Implementation details are provided in the main body.