Priority Update as a Parallel Primitive

Julian Shun[*] **Guy E. Blelloch**[*]
Jeremy T. Fineman[†] **Phillip B. Gibbons**[‡]

February 2013
CMU-CS-13-101

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[*]Carnegie Mellon University
[†]Georgetown University
[‡]Intel Labs, Pittsburgh

This work is partially supported by the National Science Foundation under grants CCF-1018188 and CCF-1218188, by Intel Labs Academic Research Office for the Parallel Algorithms for Non-Numeric Computing Program, and by the Intel Science and Technology Center for Cloud Computing.

Keywords: Memory Contention, Parallel Programming, Experimentation

Abstract

Memory contention can be a serious performance bottleneck in concurrent programs on shared-memory multicore architectures. Having all threads write to a small set of shared locations, for example, can lead to orders of magnitude loss in performance relative to all threads writing to distinct locations, or even relative to a single thread doing all the writes. Shared write access, however, can be very useful in parallel algorithms, concurrent data structures, and protocols for communicating among threads.

In this paper we study the "priority update" operation as a useful primitive for limiting write contention in parallel and concurrent programs. A *priority update* takes as arguments a memory location, a new value, and a comparison function $>_p$ that enforces a partial order over values. The operation atomically compares the new value with the current value in the memory location, and writes the new value only if it has *higher priority* according to $>_p$. On the implementation side, we show that if implemented appropriately, priority updates greatly reduce memory contention over standard writes or other atomic operations when locations have a high degree of sharing. This is shown both experimentally and with theoretical justification. On the application side, we describe several uses of priority updates for implementing parallel algorithms and concurrent data structures, often in a way that is deterministic, guarantees progress, and avoids serial bottlenecks. We present experimental results showing that a variety of such algorithms and data structures perform well under high degrees of sharing. Given the results, we believe the priority update operation serves as a useful parallel primitive and good programming abstraction as (1) the user largely need not worry about the degree of sharing, (2) it can be used to avoid non-determinism since, in the common case when $>_p$ is a total order, priority updates commute, and (3) it has many applications to programs using shared data.

1 Introduction

When programming algorithms and applications on shared memory machines contention in accessing shared data structures is often a major source of performance problems. The problems can be particularly severe when there is a high degree of sharing of data among threads. With naive data structures the performance issues are typically due to contention over locks. Lock-free data structures alleviate the contention, but such solutions only partially solve issues of contention since even the simplest lock free shared write access to a single memory location can create severe performance problems. For example, simply having all threads write to a small set of shared locations can lead to orders of magnitude loss in performance relative to writing to distinct locations. The problem is caused by coherence protocols that require each thread to acquire the cache line in exclusive mode to update a location; this cycling of the cache line through the caches incurs significant overhead—far greater than even the cost of having a single thread perform all the writes. The performance is even worse when using operations such as a compare-and-swap to atomically update shared locations.

To avoid these issues researchers have suggested a variety of approaches to reduce the cost of memory contention. One approach is to use *contention-aware schedulers* [26, 11] that seek to avoid co-scheduling threads likely to contend for resources. For many algorithms, however, high degrees of sharing cannot be avoided via scheduling choices. A second approach is to use *hardware combining*, in which concurrent associative operations on the same memory location can be "combined" on their way through the memory system [13, 12, 9, 3]. Multiple writes to a location, for example, can be combined by dropping all but one write. No current machines, however, support hardware combining. A third approach is to use *software combining* based on techniques such as combining funnels [22] or diffracting trees [21, 8]. These approaches tend to be complicated and have significant overhead, because a single operation is implemented by multiple accesses that traverse the shared combining structure. In cases where the contending operations are (atomic) updates to a shared data structure, more recent work has shown that having a single combiner thread perform the updates greatly reduces the overheads [14, 10]. This approach, however, does not scale in general. A fourth approach *partitions* the memory among the threads such that each location (more specifically, each cache line) can be written by only a single thread. This avoids the cycling-of-cache-lines problem: Each cache line alternates between the designated writer and a set of parallel readers. Such partitioning, however, severely limits the sorts of algorithms that can be used. Finally, the *test and test-and-set* operation can be used to significantly reduce contention in some settings [20, 16, 18, 17]. While contention can still arise from multiple threads attempting to initially set the location, any subsequent thread will see the location set during its "test" and drop out without performing a test-and-set. This operation has limited applicability, however, so our aim is to identify a more generally applicable operation with the same contention-reducing benefits.

Priority Update. In this paper we study a generalization of the test-and-set operation, which we call *priority update*. A *priority update* takes as arguments a memory location, a new value, and a $>_p$ function that enforces a partial order over values. The operation atomically compares the new value with the current value in the memory location, and writes the new value only if it has *higher priority* according to $>_p$. At any (quiescent) time a location will contain the highest priority value written to it so far. A test-and-set is a special case of priority update over two values, where

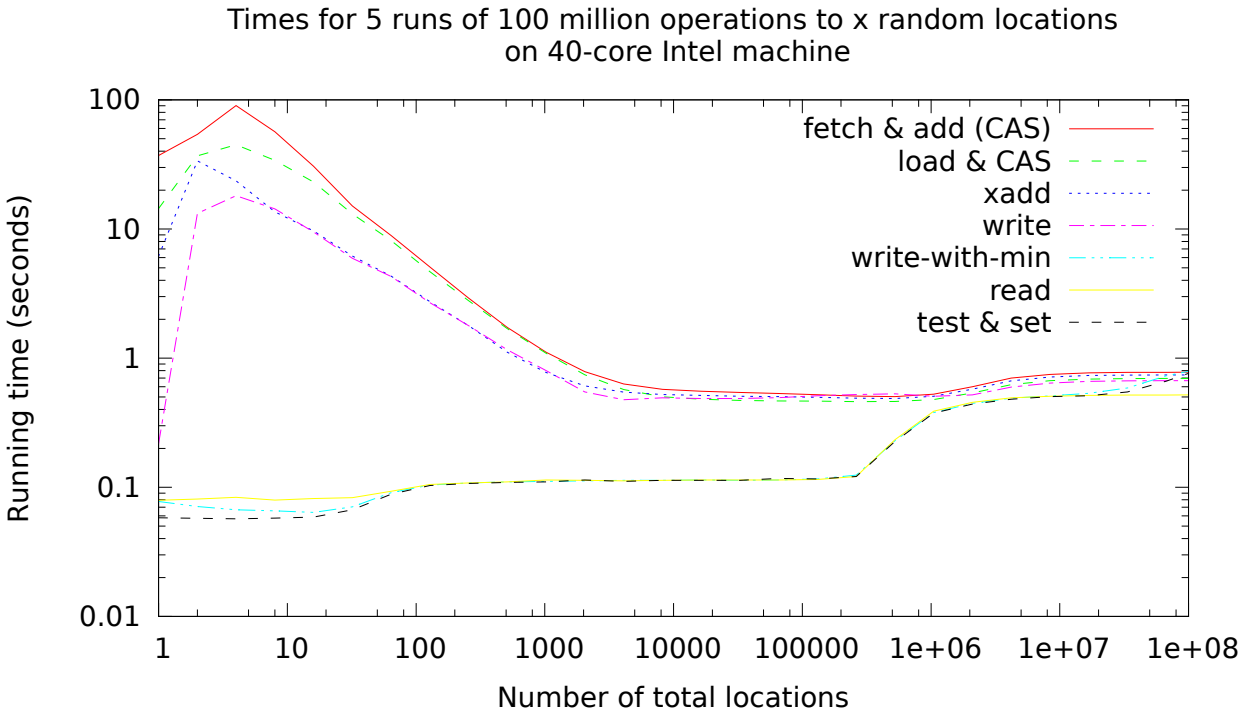


Figure 1: Times for performing different operations on a 40-core Intel Nehalem (log-log scale). Since the number of operations is fixed, fewer locations implies more operations sharing those locations.

the location initially holds 0, the new value to be written is 1, and 1 has a higher priority than 0. Another special case is the *write-with-min* over integer values, where the minimum value written has priority.

We provide evidence that the priority update operation serves as a good abstraction for programmers of shared memory machines since it is a useful in many applications on shared data (often in a way that is deterministic, guarantees progress, and avoids serial bottlenecks), and when implemented appropriately (see below) performs reasonably well under any degree of sharing. This latter point is illustrated in Figure 1. Each data point represents the time for 5 runs of 10^8 operations each on a 40-core machine. The x-axis gives the number of distinct locations being operated on—hence the leftmost point is when all operations are on the same location and at the right the graph approaches no sharing. (More details on the setup and further experimental comparisons are described in Section 3.2). As can be seen, when there is a high degree of sharing (e.g., only 8 locations) the read, the test-and-set, and the write-with-min (with random values) are all over two orders of magnitude faster than the other operations. One would expect the read to do well because the cache lines can be shared. Similarly the test-and-set does well because it can be implemented using a test and test-and-set (as described above) so that under a high degree of sharing most cache lines are again in shared mode. The steps in the curve arise each time the number of locations no longer fits within a cache at a particular level.

The write-with-min can be implemented with a read, a local comparison, and a compare-and-swap such that the compare-and-swap is needed only when the value being written is smaller than

the existing value. Thus, most invocations of write-with-min only *read* shared data, which is why the running time nearly matches the read curve. The curve shows that the high sharing case is actually the best case for a write-with-min. This implies the user need not worry about contention, although, as with reads, she might still need to worry about memory footprint and if it fits in cache.

Applications of Priority Updates. Priority updates have many applications. Here we outline several such applications and go into significantly more detail in Sections 4 and 5. The operation can be used directly within an algorithm to take the minimum or maximum of a set of values, but it also has several other important properties. Due to the fact that the operation is commutative [25, 24] (order does not matter) in the common case when $>_p$ is a total order, it can often be used to avoid non-determinism when sharing data. By assigning threads unique priorities it can also be used to guarantee (good) progress by making sure at least the highest priority thread for each location succeeds in a protocol.

Write-with-min is used in a recently introduced technique called *deterministic reservations* to implement a speculative FOR loop [1]. The idea is to give each iteration of the loop a priority based on its iteration number. Then any prefix of the iteration space can be executed in parallel such that when accessing any data that might be shared by other iterations, the iteration "reserves" the shared data using a priority update. A second "commit" phase is used to check whether the iteration has "won" on all the data it shares and if so proceeds with any updates to the global shared state. The approach has the advantage in that it guarantees the same order of execution of the iterates as in the sequential order. Furthermore, it guarantees progress since at least the earliest iterate will always succeed (and often many iterates succeed in parallel, if different locations are used).

Priority updates have applications in many graph algorithms. They can be used in parallel versions of both Boruvka's and Kruskal's minimum-spanning-forest algorithms to select minimum-weight edges. They can be used in Bellman-Ford shortest paths to update the neighbors of a vertex with the potentially shorter path. They can also be used in certain graph algorithms to guarantee determinism. In particular any conflicts can be broken using priorities so that they are always broken in the same way. For example, in breadth-first search (BFS), it can be used to deterministically generate a BFS tree.

Priority updates on locations can also be used to efficiently implement a more general dictionary-based priority update where the writes/inserts are made based on keys. For example, all writers might insert a character string into a dictionary with an associated priority. We show experimentally that using hash tables this operation is almost no more expensive than writing to a location and that as with priority updates it remains cheap, actually becomes cheaper, even under high degrees of sharing. We use the hash table to implement a prioritized remove-duplicates algorithm.

In this paper we describe these algorithms and present experimental results. We study the performance of BFS, Kruskal's MSF algorithm, and a maximal matching algorithm. We present timing results when vertices have very high degree to show the case of high sharing and, for BFS and maximal matching, we compare timings with versions that use writes instead of priority updates.

Contributions. In summary, the main contributions of this paper are as follows. First, this paper generalizes and unifies special cases of priority update operations from the literature, and is the first

```
procedure PRIORITYUPDATE(addr, newval, >_p)
    oldval ← *addr
    while (newval >_p oldval) do
        if CAS(addr, oldval, newval) then
            return
        else
            oldval ← *addr
```

Figure 2: Priority Update Implementation

to call out priority update as a key primitive in ensuring that having many threads updating a few locations does not result in cache/memory system performance problems. Second, we provide the first comprehensive experimental study of priority update vs. other widely-used operations under varying degrees of write sharing, demonstrating up to orders of magnitude differences on modern multicores from both Intel and AMD. We also present the first analytic justification for priority update's good performance. Third, we present algorithms for a number of important problems that demonstrate a variety of ways to benefit from priority updates. While several of the algorithms appear in [1] (which uses a primitive akin to write-with-min), the connected components, shortest paths, and maximal matching algorithms are new. Finally, we present the first experimental study demonstrating the (good) performance of priority update algorithms on inputs that result in a high degree of write sharing, extending the study in [1] by considering a wider range of degrees of sharing, running on more cores, providing a comparison versus using regular writes, and including maximal matching among the problems studied.

2 Priority Updates

A ***priority update*** takes as arguments a memory location containing a value of type T, a new value of type T to write, and a binary comparison function $>_p : T \times T \rightarrow bool$ that enforces a partial order over values. The priority update atomically compares the two values and replaces the current value with the new value if the new value has higher priority according to $>_p$. It does not return a value. In the simplest form, called a ***write-with-min*** (or write-with-max), T is a number type, and the comparison function is standard numeric less-than (or greater-than). Our implementation below, however, allows T to be an arbitrary type with an arbitrary comparison function. When $>_p$ defines a total order over T, priority updates commute—i.e., whatever the ordering of the updates, the value left in the location at the end will be the same.

A priority update can be implemented as shown in Figure 2 using a compare-and-swap (CAS). Because CAS (on a single word, or sometimes a double length word) is provided as a hardware atomic on modern machines, no new hardware primitives are required. If the value does not "fit" in a word, one can use a pointer to the actual data being compared (pointers certainly fit in a word), so the implementation can easily be applied to a variety of types (e.g., structures with one of the fields being compared, variable-length character strings with lexicographic comparison, or even complex structures with an NP-hard comparison). One should distinguish the comparison function $>_p$ defining the partial order over the values from the "compare" in compare-and-swap, which is a comparison for equality and is applied to the indirect representation of the value (e.g., the bits in the pointer) and not the abstract type. We assume the object is not mutated during the operation so

that equality of the indirect representation (pointer) implies equality of the abstract value.

In the best case, the given implementation of priority update completes immediately after a single application of the comparison function, determining that the value already stored in the location has higher priority than the new value. Otherwise an ***update attempt*** occurs with the compare-and-swap operation. (Because our implementation uses CAS to attempt an update, we will also refer to this as a ***CAS attempt***.) If successful, we say that an ***update*** occurs. If not, the priority update retries, completing only when the value currently stored has an equal or higher priority than the new value, or when a successful update occurs.

As noted earlier, a test-and-set is a special case of priority update over two values. A ***write-once*** operation is another special case of a priority update where the contents of a location starts in an "empty" state and once one value is written to the location, making it "full", no future values will overwrite it. As with test-and-set there are just two priorities—empty and full. A third special case is the priority write from the PRAM literature [15]—a synchronous concurrent write from the processors that resolves writes to a common location by taking the value from the highest (or lowest) numbered processor. This can be implemented by using pointers to (processor number, value) pairs: *addr* contains a pointer to the current pair, *newval* is a pointer to a new pair, and $>_p$ chases the two pointers and compares the processor numbers. We note that both test-and-sets and PRAM-style priority writes each commute because the values form a total order, but that write-once operations do not because there are many values with equal priority and the first one that arrives is written.

3 Contention in Shared Memory Operations

In this paper we distinguish between sharing and contention. By ***sharing*** we mean operations that share the same memory location (or possibly other resource)—for example, a set of instructions reading a single location. By ***contention*** we mean some form of sequential access to a resource that causes a bottleneck. Contention can be a major source of performance problems on parallel systems while sharing need not be. A key motivation for the priority update operation is to reduce contention under a high degree of sharing.

Although contention can be a problem in any system with sequential access to a shared resource, the problem is amplified for memory updates on cache coherent shared memory machines because of the need to acquire a cache line in exclusive mode. In the widely used MESI (Modified, Exclusive, Shared, Invalid) protocol [19] and its variants, a read can acquire a cache line in shared mode and any number of other caches can simultaneously acquire the line. Concurrent reads to shared locations therefore tend to be reasonably efficient. In fact since most machines support some form of snooping, reading a value that is in another cache can be faster than reading from memory.

On the other hand, in the MESI protocol (and other similar protocols implemented on current multicore machines) concurrent writes can be very inefficient. In particular the protocol requires that a cache line be acquired in exclusive mode before making an update to a memory location. This involves invalidating all copies in other caches and waiting for the invalidates to complete. If a set of caches simultaneously make an update request for a location (or even different locations within a line) then the cache line will need to be acquired in exclusive mode by the caches one at a time, doing a dance around the machine. The cost of each acquisition is high since it involves

communicating with the cache that has the line in exclusive or modified state, waiting for it to complete its operation, getting a copy of the newly updated line, and updating any tables that keep track of ownership. If the cores make a sequence of requests to a small set of locations then all requests could be rotating through the caches. Because of the cost of the protocol, this can be significantly more expensive than simply having one core do all the writes. On a system with even 8 cores this can be a serious performance bottleneck, and on one with 40 cores it can be crippling, as the experiments later in this section indicate.

If there are a mix of read and write requests to a shared location then the efficiency will fall between the all-read and all-write cases, depending on the ratio of reads to writes as well as more specifics about how the protocol is implemented. As shown later in the section, for this case there is actually a significant difference in performance between the protocols implemented on the AMD Opteron and the Intel Nehalem multicores.

In this section we study the cost of write sharing among caches (cores) on modern multicores. Along with other operations, we study the cost of a priority update and give both theoretical and experimental justification of why it is reasonably efficient.

3.1 Performance Guarantees for Priority Update

As discussed in Section 2, the operation is a further generalization of the test-and-set and write-once operations. Unlike those operations, in a priority update a value can change multiple times instead of just once. However, if the ordering of operations is randomized then the number of updates is small and as with test-and-set and write-once most invocations only read shared data.

In this section we consider priority update operations where $>_p$ defines a total order over the value domain T. Values can be repeated, so that the number of operations n can be much larger than the number of priorities or the size of T. We say that a collection of priority update operations has ϕ ***occurring priorities*** if the values in those operations fall into exactly ϕ distinct priorities according to $>_p$.

We begin with the simplest case of a sequence of priority updates, performed in random order. Here, all update attempts succeed as there are no concurrent CAS operations. This simple lemma shows that the value stored in the location is updated very few times.

Lemma 1 *Consider a random sequential ordering on a collection of priority update operations to a single location, with ϕ occurring priorities. Then H_ϕ updates occur in expectation and $O(\ln \phi)$ updates occur with high probability (in ϕ), where $H_i \approx \ln i$ is the ith harmonic number.*

Proof. Let S be the subsequence of priority updates that are the first occurrences in the original sequence of a distinct priority—these are the only operations that could possibly perform an update. Let X_k be an indicator for the event that the kth operation in S performs an update. Then $X_k = 1$ with probability $\frac{1}{k}$, as it updates only if its priority is the highest among the first k operations in S. The expected number of updates is then given by $E[X_1 + \cdots + X_\phi] = E[X_1] + \cdots + E[X_\phi] = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{\phi} = H_\phi$. Chernoff bounds give a high probability result. □

We generalize Lemma 1 to provide bounds on the runtime when performing priority updates in parallel under two models. In either model, we assume that if multiple concurrent CAS'es are executing an update attempt, the one that "wins" and successfully updates the value is independent of the data being written.

We assume that each priority update has a random distinct value. The randomness is needed in our analysis to avoid pessimal cases such as when the values arrive in increasing priority order (so that *all* operations would execute CAS attempts). The distinctness assumption provides an upper bound on the general case, and the bounds can be readily sharpened to take into account the actual number of occurring priorities, as in Lemma 1. Without loss of generality, then, we can assume each value is its relative rank in the total order; thus, the values in a collection of n priority updates form the set $\{1, \ldots, n\}$, with 1 being the highest priority and each location initialized to a special lowest-priority value ∞.

Our models are based around a simplified cache-coherence protocol, where a cache line can be in invalid, shared, or exclusive mode. A core performing a CAS requests the relevant cache line in exclusive mode, thereby invalidating the line in all other caches, and performs the CAS.[1] When reading a cache line that is invalid in the local cache, the core first requests the line in shared mode then performs the read. We charge a constant cost of c for acquiring the line in either mode, but some acquisitions may serialize due to conflicts depending on which model we adopt.

In the ***fair model***, we view outstanding cache-line requests to a particular memory location as ordered in a queue. New requests must be added to the end of the queue. When a CAS (exclusive request) is serviced, no other operations may proceed. When a read is processed, all other reads before the next CAS in the queue may be serviced in parallel, and if the cache line is modified, a cost of c is charged for acquiring the line (the first reader puts the line in shared mode).

In the ***adversarial model***, operations are not queued. Instead, an adversary may order any outstanding CAS and read operations arbitrarily (e.g., based on location being written), but without considering the actual value being written. Bounds for the adversarial model appear in Appendix A.1.

To analyze priority updates to a single location in the fair model, view operations as being processed in rounds due to the queue ordering. Each round processes p operations, one from each core, which may be either of the two steps of a priority update: a read or a CAS.[2] More precisely, let v_j denote the value stored at the start of round j. For any core performing the read step, we pessimistically assume that it observes the value v_j. The core then compares its value against v_j, and commits to either performing a CAS in round $j + 1$ or skipping the CAS attempt step and proceeding to the next operation (i.e., issuing another read). Since a CAS in round $j + 1$ is based on the value observed in round j, there is at most 1 successful CAS per round. All reads between consecutive CAS attempts complete in c time, so we can charge those reads against the preceding CAS attempt. The main goal is to bound the number of unsuccessful CAS attempts.

We have $v_1 = \infty$ initially. Every core issues a read in round 1, compares its value against ∞, and then issues a CAS in round 2 comparing against $v_1 = \infty$. Because the CAS attempts are serialized, the time to complete round 2 is $\Theta(cp)$. Exactly one core (the first one in the queue) succeeds in round 2, so the value v_3 observed at the start of round 3 is one drawn uniformly at random from $\{1, \ldots, n\}$.

[1] To clarify, once a core is granted exclusive mode, our model assumes that the CAS completes immediately. A priority update, however, consists of two steps—a read and a CAS—and the line may be invalidated in between those two steps. These assumptions are supported by our experiments on both Intel Nehalem and AMD Opteron multicores.

[2] Here, we assume the type fits in a word. The analysis readily extends to the more general case where $>_p$ must chase pointers.

Lemma 2 *The total cost for performing n priority updates (with random distinct values) to a single location using p cores under the fair model is $O(\frac{n}{p} + c \ln n + cp)$ in expectation.*

Proof. By Lemma 1, there are $O(\ln n)$ successful updates, so the question is how many unsuccessful CAS attempts there are. We start by bounding the number of priority updates that include at least 1 failed CAS.

An unsuccessful CAS occurs only if a successful CAS is made in the same or preceding round (which is bounded by $O(\ln n)$ in Lemma 1). We call ***phase*** i the set of rounds during which a) the value stored in the location falls between $\frac{n}{2^{i-1}}$ and $\frac{n}{2^i}$, and b) a successful CAS occurs. We wish to bound the number of new priority updates during these rounds that perform a (failed) CAS attempt. First, observe that phase i consists of $O(1)$ rounds in expectation, as each successful update has probability $\frac{1}{2}$ of reducing the value below the threshold of $\frac{n}{2^i}$. Moreover, in each of these rounds, each core has probability at most $\frac{1}{2^{i-1}}$ of performing a priority update of a value below $\frac{n}{2^{i-1}}$. Summing across all cores and all rounds in the phase, the expected number of (failed) priority updates during phase i is at most $O(\frac{p}{2^{i-1}})$. Summing across all phases, the total number of such failed priority updates is $O(p)$.

A failed priority update may retry several times, but a random failed update has probability $\frac{1}{2}$ of retrying through each subsequent phase since the value stored at the location is halved. Thus, there are an expected $O(1)$ retries per priority update that make any CAS attempt. Combining with the above, we get $O(p)$ unsuccessful CAS attempts.

We charge c for each of the $O(\ln n)$ successful and $O(p)$ unsuccessful CAS'es. As for the reads, any of the reads that must reacquire a cache line (and hence cost c) can be charged against the preceding CAS attempt, only doubling the cost, the first read costs c, and the remaining reads and all local computation costs $O(\frac{n}{p})$, completing the proof. □

The above results are for performing priority updates to a single location. Now we analyze the cost for *multiple locations* where cores apply operations to locations chosen uniformly at random from $\{1, \ldots, m\}$, where m is the number of locations. Let n_i be the number of operations at the ith location. Here, we assume that all locations can fit simultaneously in cache and that there are no false-sharing effects. The difficulty here is that the round analysis only applies to each location—the model has a separate queue for each location, and simply multiplying the CAS-components of the bound by m is too pessimistic.

Theorem 3 *The total cost for performing n priority updates (with random distinct values) to m randomly chosen locations under the fair model is $O(\frac{n}{p} + cm \ln(\frac{n}{m}) + (cp)^2)$ in expectation.*

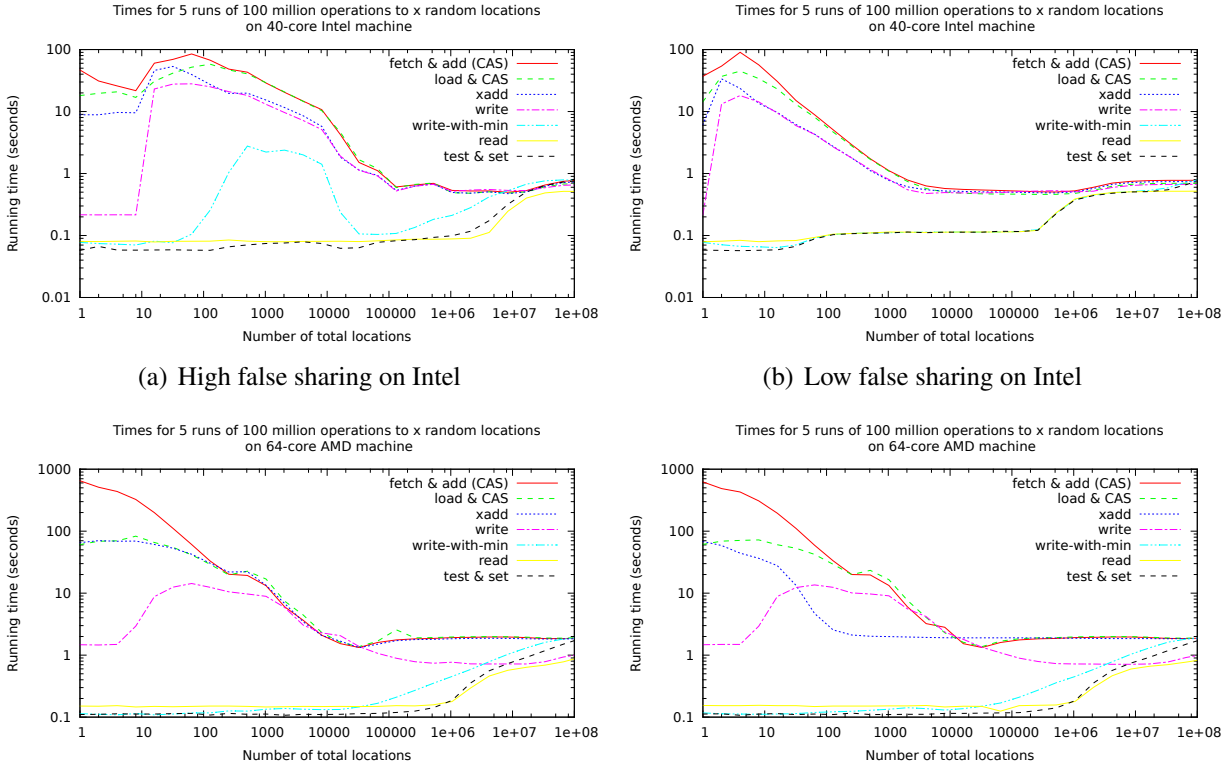
Proof. The analysis of Lemma 2 says that there are at most $O(\ln(n_i) + p)$ CAS attempts when p cores perform $O(n_i)$ updates to location i. Increasing the number of locations only decreases the number of CAS failures, since not all cores choose the same location. So a bound of $O(\frac{n}{p} + cm \ln(\frac{n}{m}) + cmp)$ follows by maximizing the logarithmic term (setting $n_i = \frac{n}{m}$ for all i) and multiplying by m locations. As noted above, this bound is pessimistic, so we will improve it for $m \geq p$. The $O(cm \ln(\frac{n}{m}))$ term seems inherent because each update invalidates the line in all other caches, so the work to reload those lines later is $O(cmp \ln(\frac{n}{m}))$ (which is divided across p cores)—our goal is to reduce the $O(cmp)$ term.

Consider the round analysis as in Lemma 2 applied to a single location. The main question is how many (unsuccessful) CAS'es are launched on that location during a round containing a

successful CAS. The maximum length of a round is $O(cp)$ if every core performs a CAS attempt. Each core may thus sample up to $O(cp)$ locations within a round (each sample is independent from the rest), giving a probability of $O(\frac{cp}{m})$ of choosing this location in any of those attempts. Summing across all cores, the expected number of priority updates to this location in each round is $O(\frac{cp^2}{m})$, only some of which my actually perform a CAS attempt. As in Lemma 2, the likelihood of performing a CAS attempt decreases geometrically in each phase. So the total number of failed CAS'es on this location is $O(\frac{cp^2}{m})$. Summing across all locations gives $O(cp^2)$ failed attempts, each with cost c. □

For reasonably sized n, the bounds in this section are much better than the bounds for operations that always have to access a cache line in exclusive mode. Such operations will run in $O(cn)$ at best assuming either the fair or adversarial model—all accesses will be sequentialized and will involve a cache miss.

3.2 Experimental Measurements of Contention



(a) High false sharing on Intel

(b) Low false sharing on Intel

(c) High false sharing on AMD

(d) Low false sharing on AMD

Figure 3: Times for performing a set of operations to varying numbers of memory locations with high and low degrees of false sharing (log-log scale). Since the number of operations is fixed, fewer locations implies more operations sharing those locations.

We now look at the cost of contention on two contemporary shared memory multicores (from Intel and AMD) for a variety of memory operations—write-with-min, test-and-set, fetch-and-add using CAS, fetch-and-add using the x86 assembly instruction `xadd`, load-and-CAS, (plain) write,

and read[3]. The Intel machine is a 40-core (with hyper-threading) Nehalem-EX machine with 4×2.4GHZ Intel 10-core E7-8870 Xeon processors, a 1066MHz bus, and 256GB of main memory. The AMD machine is a 64-core machine with 4×2.4GHZ AMD 16-core Opteron 6278 processors. The programs on the Intel machine were compiled with Intel's `icpc` compiler (version 12.1.0, which supports Cilk Plus) with the `-O3` flag. The programs on the AMD machine were compiled using the `g++` compiler (version 4.8.0 which supports Cilk Plus) with the `-O2` flag.

In the experiments, we perform 10^8 operations on a varying number of random locations. On each machine we performed two sets of experiments. The first set of experiments choose the locations randomly in $[0, x)$ where x is the total number of locations written to and locations 0 through x appear contiguously in memory. The second set of experiments choose the locations randomly from $\{h(i) : i \in [0, x)\}$ where $h(i)$ is a hash function that maps i to a value in $[0, 10^8)$. In the first set of experiments, there will be high false sharing due to concurrent writing to locations on the same cache line. The second set is supposed to represent what we view as a more common usage of priority update, which is a set of writes to a potentially large set of locations but for which there is heavy load at a few locations. There is significantly less effect of false sharing in the second set since the heavily loaded locations are unlikely to be on the same cache line.

Figure 3(a) shows that with high sharing (low number of total locations) and high false sharing, write-with-min outperforms plain write, both versions of fetch-and-add, and load-and-CAS by orders of magnitude. Due to an Intel anomaly (see Appendix A.2), there is a spike in the running time for write-with-min between 256 and 8192 locations, but even with this anomaly write-with-min still outperforms plain write, fetch-and-add and load-and-CAS by an order of magnitude. This anomaly disappears when we reduce the false sharing effect, as shown in Figure 3(b). Figure 3(b), which is a repeat of Figure 1, also shows that the performance of write-with-min is very close to the performance of both test-and-set and read. For writing to 10^8 locations (the lowest degree of sharing), write-with-min is slightly slower than fetch-and-add, and test-and-set is slightly slower than write (even though intuitively fetch-and-add does more work than write-with-min and write does more work than test-and-set). We conjecture this behavior to be due to the branch in write-with-min and test-and-set obstructing speculation on the hardware compare-and-swap instruction. We note that `xadd` is consistently faster than implementing a fetch-and-add with a CAS, since the CAS could fail. Also, we noticed that `xadd` performs about the same as a CAS without a load. Preliminary experiments on a new 32-core Intel Sandy Bridge machine yielded results that were qualitatively similar to Figures 3(a) and 3(b).

Figures 3(c) and 3(d) show the same two experiments on the AMD machine. Note that even with high false sharing, the anomaly for the write-with-min operation observed for the Intel machine does not appear for the AMD machine. Except for the write-with-min anomaly, the performance on the Intel machine is better than the performance on the AMD machine.

4 Applications of Priority Update

Priority updates are well-suited to a widely applicable two-phase programming style, which we call *update-and-read* and *reserve-and-commit*. Reserve-and-commit is a special case of update-and-read but with a specific type of usage. An ***update-and-read*** program alternates two types of

[3]The read includes a write to local memory to get around compiler optimizations.

phases. During an *update* phase, multiple update attempts occur on some collection of objects, using either a priority update, a plain write, or another write primitive. During the subsequent *read* phase, the value that was successfully recorded is read. Using priority updates or write-once operations during the update phase is desirable to achieve better performance (see Section 5). Moreover, the commutative nature of priority updates implies that the values stored at completion of the read phase are deterministic.

When operating on a collection of interacting objects (e.g., vertices of a graph), where each object seeks to update a "neighborhood" of objects, a ***reserve-and-commit*** style is more appropriate. In the *reserve* (update) phase, each object in parallel attempts to reserve the neighborhood of objects it would modify. In the *commit* (read) phase, each object in parallel checks whether it holds a reservation on its neighborhood, and if so, performs the desired modifications. There should be a synchronization point between the reserve and commit phases, guaranteeing that commits and reserves cannot occur concurrently with each other. Since reservations are exclusive (indeed reservations are acting as mutual-exclusion locks), this approach guarantees that each commit behaves atomically. As with the generic update-and-read, the reservations can be implemented using either a priority update or a plain write. The former is more desirable both for performance and to guarantee forward progress when multiple objects are reserved.

If used correctly and employing a priority update, this reserve-and-commit style can be thought of as a special case of transactional programming, but one in which forward progress guarantees are possible. The reserve phase essentially speculatively attempts a "transaction," and the commit phase commits transactions that do not interfere. By using priority updates, there is a total order over reservations, guaranteeing that at least one reserver (i.e., the one with highest priority) is able to commit. This forward-progress guarantee does not apply when using a plain write or a write-once, as it is possible that no reserver "wins" on all of its neighbors.

The technique of ***deterministic reservations*** [1] extends this reserve-and-commit abstraction to an entire parallel loop. In this technique, a sequence of iterates (typically randomly ordered) are considered in parallel, and the reservations are made using priority updates according to the iterates' ranks in the sequence. In the commit phase, iterates that successfully reserved their neighborhoods perform their commit operation. All uncommitted iterates are gathered, and this process repeats until no iterates remain. Deterministic reservations has several appealing features [1]. First, the behavior is consistent with a sequential execution of the loop, so the results are deterministic. Second, the performance can be tuned by operating on a prefix of the sequence: a smaller prefix decreases data sharing thereby decreasing total work, whereas a larger prefix may allow more parallelism [2]. Third, since the reservations themselves are based on iterate priorities, some forward progress is guaranteed in each round.

Note that because the highest priority update succeeds for *each* location, priority updates often enable considerable *parallel* progress in each update-and-read phase, yielding good parallel speedups (see Section 5). For example, with deterministic reservations, often $\Omega(p)$ iterates succeed in parallel.

The remainder of this section describes several algorithms that use priority update, most of which employ some form of update-and-read. The exceptions are connected components and shortest paths, where a write-with-min is used to asynchronously update values. In some of these

cases (e.g., breadth-first-search and maximal matching), several write primitives maintain correctness of the algorithms and priority updates are just desirable for performance. In others (e.g., connected components, minimum spanning forest, and shortest paths), the priority update is necessary for correctness of the given algorithm.

Breadth-First Search. The ***breadth-first search*** (BFS) problem takes as input an undirected graph G and a source vertex s and returns a breadth-first-search tree represented by an array of parent pointers. The BFS algorithm proceeds in rounds, during which all vertices on the frontier (initialized to contain only the source vertex) attempt to place all of their neighbors on the next frontier. Our experiments use the BFS implementation from the publicly available problem-based benchmark suite (PBBS) [23, 1]. To guarantee that each vertex is added only once, each round is implemented with an update-and-read style. During the update phase, a frontier vertex writes its ID to its neighbors. During the read phase, each vertex counts its reserved neighbors. The next frontier-vertex array is then constructed by using prefix sums over these counts to determine where each vertex's reserved neighbors should begin in the output array and then putting them in the appropriate array slots.

This BFS algorithm may be correctly implemented by using priority updates, write-once, or plain writes, with plain writes being less efficient (see Section 5) and priority updates guaranteeing a deterministic BFS-tree output [1].

Maximal Matching. The ***maximal matching*** (MM) problem takes as input an undirected graph $G = (V, E)$ and returns a subset $E' \subseteq E$ such that no two edges in E have any endpoint in common (matching) and all edges not in E' share at least one endpoint with some edge in E' (maximal). Our simple algorithm for solving the MM problem uses a reserve-and-commit style. We maintain a size $|V|$ array of boolean flags, indicating whether a vertex has been matched yet (the array is initialized to all *false*). During each iteration of the algorithm all edges remaining in the graph are processed in multiple logical phases: 1) every edge checks if either of its endpoints is matched (flag set to *true*); if so, the edge removes itself from the graph, 2) all remaining edges reserve both endpoints by using a write-with-min with the edge's unique ID, and 3) every remaining edge checks if both of its endpoints contain its reservation; if so, it joins the matching, removes itself from the graph, and sets both of its endpoints' flags to *true*. In our implementation, we merge phases 1 and 2 into a single phase, reducing the amount of inter-phase synchronization. The algorithm can also be implemented using write-once or plain writes, but forward progress is not guaranteed because no edge may succeed in reserving both its endpoints.

Connected Components. For an undirected input graph $G = (V, E)$, a connected component $C \subseteq V$ is one in which all vertices in C can reach one another. The ***connected components*** problem is to find $C_1, \ldots, C_k$ such that each C_i is a connected component, there is no path between vertices belonging to different components, and $C_1 \cup \cdots \cup C_k = V$. Our simple vertex-based algorithm assigns each vertex a unique identifier at the start, and in each iteration every vertex sets its ID to the minimum ID of all its neighbors. The algorithm terminates when no vertex's ID changes in an iteration. In each iteration, each vertex performs a write-with-min to all of its neighbors' IDs. This is an example of using write-with-min to guarantee the correctness of an algorithm, and where the priority update yields a remarkably simple solution.

Minimum Spanning Forest. For an undirected graph $G = (V, E)$ the ***spanning forest*** problem returns a set of edges $F \subseteq E$ such that for each connected component $C_i = (V_i, E_i)$ of G, a spanning tree of C_i is contained in F and F contains no cycles. The ***minimum spanning forest*** (MSF) problem takes as input an undirected graph $G = (V, E)$ with weights $w : E \to \mathbb{R}$ and returns a spanning forest of minimum total weight.

Most MSF algorithms begin with an empty spanning forest and grow the spanning forest incrementally by adding "safe" edges (those with minimum weight crossing a cut). See, e.g., [6] for details. Kruskal's algorithm considers edges in sorted order by weight and iteratively adds edges that connect two formerly unconnected components, using a union-find data structure to query the components. This algorithm can be parallelized by accepting an edge into the MSF if no earlier edge in the sorted order is connected to the same component. Boruvka's algorithm is similar to Kruskal's except that Kruskal's sorts all edges initially and employs a union-find data structure over connected components, whereas Boruvka's algorithm uses contraction to reduce connected components.

In either case, an update-and-read style is applicable: During the update phase each edge writes its weight/ID to components containing each endpoint, and during the read phase the ID read from each component is added to the MSF. As with connected components, the write-with-min is required for correctness here, otherwise, the edge added may not be a safe edge. For our experiments, we use the parallel implementation of Kruskal's algorithm from PBBS [23, 1].

Shortest Paths. The ***single-source shortest paths*** problem takes as input a graph $G = (V, E)$ and a source s, and if G does not contain a negative cycle, outputs the shortest distance from s to every other vertex in V. The *Bellman-Ford* algorithm (see, e.g., [6]) solves the single-source shortest paths problem in the presence of negative weights. Our algorithm is a simple parallel version of Bellman-Ford. We maintain a $ShortestPaths$ vector, initialized all to ∞ except for the source vertex whose entry is 0. In each iteration every frontier vertex attempts to pass its $ShortestPaths$ value plus the edge length to each of its neighbors; we use a write-with-min on the $ShortestPaths$ entry of the neighbor in order to allow updates to occur in parallel. As in the sequential algorithm [6], we iterate until the $ShortestPaths$ values converge, and if they do not converge after $|V|$ iterations, then the graph contains a negative cycle (and the algorithm reports it).

This is another case where a write-with-min is necessary for correctness. If an unprotected write is employed, the *ShortestPaths* value is not guaranteed to decrease in each iteration, and hence the values may not converge in $|V|$ rounds even without negative-weight cycles.

Other Applications. Priority updates can be used to implement a dictionary data structure that supports insertions of key/value pairs where values are combined with a priority update on insertion of duplicate keys. This application is described in more detail in Appendix A.3.

Priority updates are applicable to other problems whose solutions are implemented using deterministic reservations (an extension of reserve-and-commit) [1]. These problems include Delaunay triangulation/refinement [7], maximal independent set, and randomly permuting an array. In most of these cases (as with maximal matching), write-once implementations are correct, but because multiple reservations are required to commit, priority updates are necessary to guarantee forward progress. Moreover, the priority update version guarantees a consistent, deterministic output once

Input	Num. Vertices	Num. Edges	Sharing Level
3D-grid	10^7	3×10^7	Low
random-local	10^7	5×10^7	Low
rMat	2^{24}	5×10^7	Medium
comb	10^7	2×10^7	High
exponential	5×10^6	1.1×10^8	High
star	5×10^7	5×10^7	High

Table 1: Inputs for graph applications

the random numbers are fixed. Priority updates are useful in other parallel algorithms that, like deterministic reservations, impose a random priority order among elements [4].

5 Experiment Study: Applications

We used the Intel Nehalem machine set-up described in Section 3.2. For sequential programs, we used the `g++` 4.4.1 compiler with the `-O2` flag.

For the breadth-first search, maximal matching, remove duplicates, and minimum spanning forest applications, we run experiments on inputs that exhibit varying degrees of sharing. For the first three, we compare our implementations using write-with-mins with an alterative implementation using plain writes. (Recall that plain writes cannot be used in our MSF algorithm.) We describe the experimental setup for each of applications in more detail below. All times reported are based on the median of three trials.

The inputs used for the graph algorithms are shown in Table 1. Because in our algorithms a vertex can only be simultaneously processed by its neighbors, graphs with low degree overall exhibit low sharing while graphs containing some vertices of high degree can exhibit high sharing (depending on the application). ***3D-grid*** is a grid graph in 3-dimensional space. Every vertex has six edges, each connecting it to its two neighbors in each dimension, and thus is a low-sharing graph. ***random-local*** is another low-sharing graph in which every vertex has five edges to neighbors chosen randomly where the probability of an edge between two vertices is inversely correlated with their distance in the vertex array (vertices tend to have edges to other vertices that are close in memory). The ***rMat*** graph is a graph with a power-law distribution of degrees. We used the algorithm described in [5] with parameters $a = 0.5, b = c = 0.1, d = 0.3$ to generate the rMat graph. The ***comb*** graph is a three layered graph (see Figure 4) with the first layer containing only the source vertex r, second layer containing $n - k - 1$ vertices and third layer containing k vertices. The source vertex has an edge to all vertices in the second layer, and each vertex in the second layer has an edge to a randomly chosen vertex in the third layer. There are a total of $4(n - k - 1)$ directed edges in this graph. For our experiments we used varying k to model concurrent writes to k random locations. The ***exponential*** graph has an exponential distribution in vertex degrees, and given a degree, incident edges from each vertex are chosen uniformly at random. The ***star*** graph is a graph with four "center" vertices and edges connecting each of the $n - 4$ remaining vertices to a randomly chosen center vertex (total of $n - 4$ edges).

In BFS, since many vertices may compete to place the same neighbor on the next frontier,

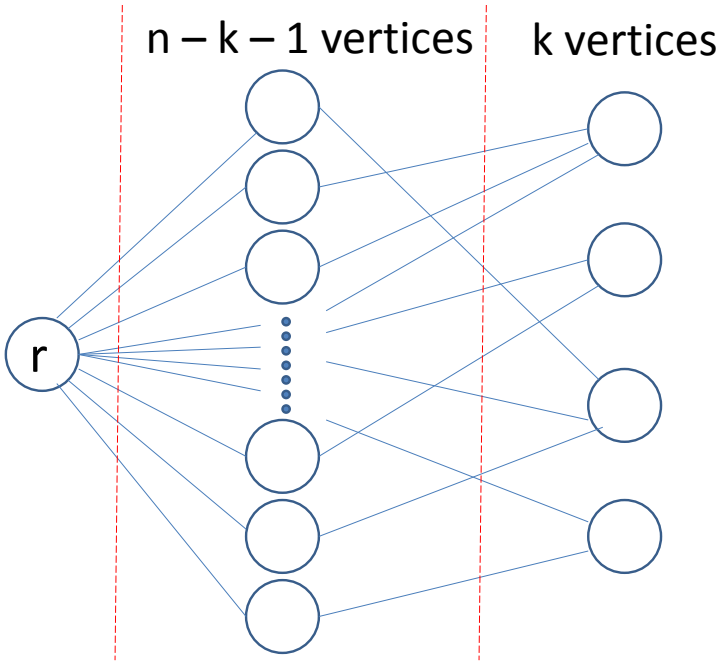


Figure 4: Comb graph (used for BFS experiments)

there can be high sharing. The comb graph illustrates this: In the first round the source vertex r explores the $n - k - 1$ vertices in the second level, without sharing; in the second round all of the second level vertices contend on vertices in the third level (see Figure 4). We chose to model sharing on comb graphs with different k values in order to observe the effect of write sharing that we discussed in Section 3, where a lower k value corresponds to higher sharing. We show two versions of BFS, one which uses a write-with-min (***writeMinBFS***) as the atomic operation to place neighbors on the frontier, and the other which uses a plain write (***writeBFS***). Figure 5 compares the two BFS implementations and the sequential BFS implementation (***serialBFS***) as a function of number of cores on the comb graph for $k = 4$. Table 1(a) shows the running times for each of the BFS implementations on all of our graphs. Our results show that the write-with-min implementation performs well even on high-sharing graphs while the plain-write implementation does poorly on high-sharing graphs (barely outperforming serialBFS). Using a family of comb graphs with varying k we model the effect of write sharing on k locations for BFS when utilizing all 40 cores. A lower value of k corresponds to higher sharing. Figure 6 shows that for values of k up to around 5000, writeMinBFS outperforms writeBFS, by nearly an order of magnitude for small k. For higher values of k where there is little sharing, writeMinBFS is slower than writeBFS due to the overhead of the test and compare-and-swap, however using these operations makes writeMinBFS deterministic. For values of k less than 50 (very high sharing), writeBFS is worse than even the sequential implementation (serialBFS).

In each iteration of maximal matching, each edge reserves its two endpoints if the endpoints have not yet been matched. The star graph causes high sharing because many edges will attempt to reserve the same "center" vertex simultaneously. We use four center vertices instead of one

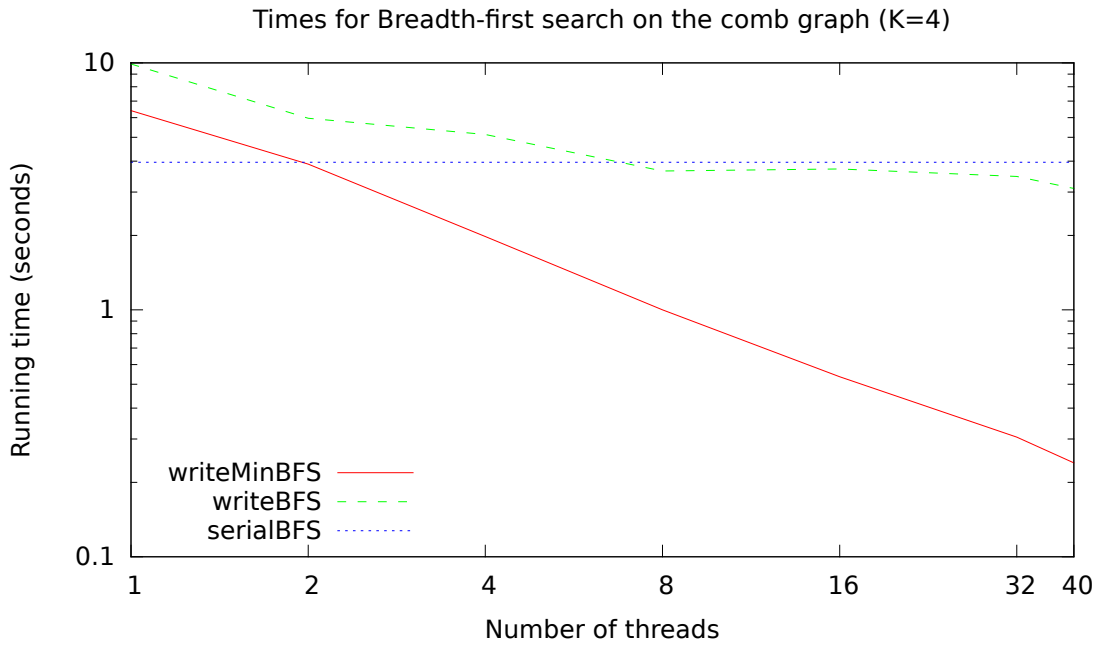


Figure 5: BFS times vs. number of cores on the comb graph for $k = 4$ (log-log scale)

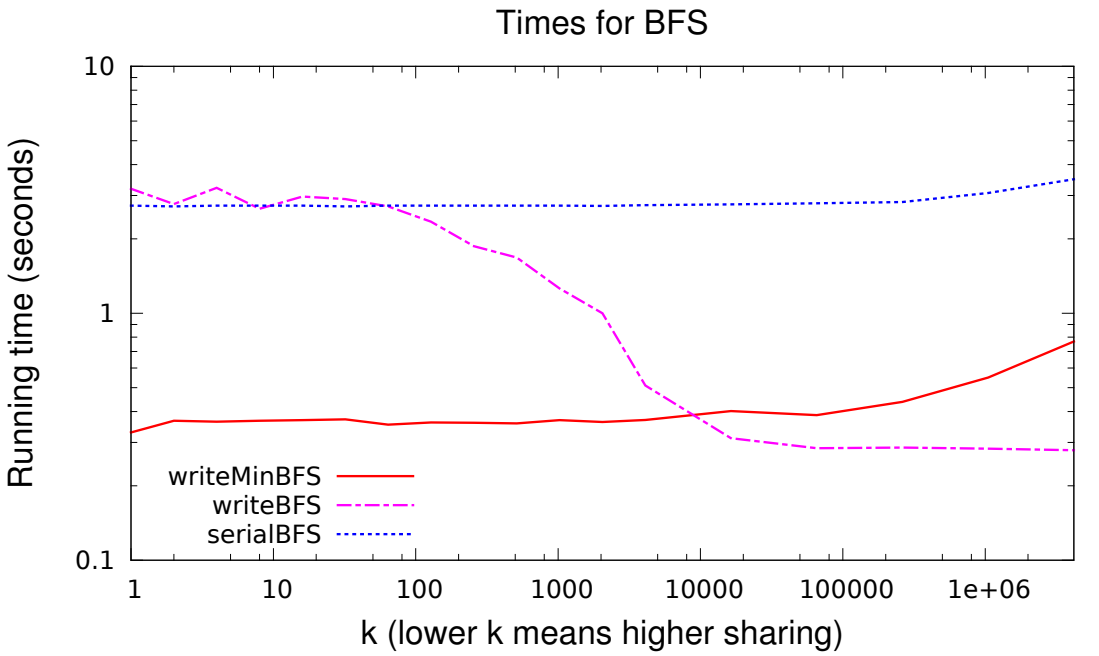


Figure 6: BFS times on different comb graphs using 40 cores (log-log scale)

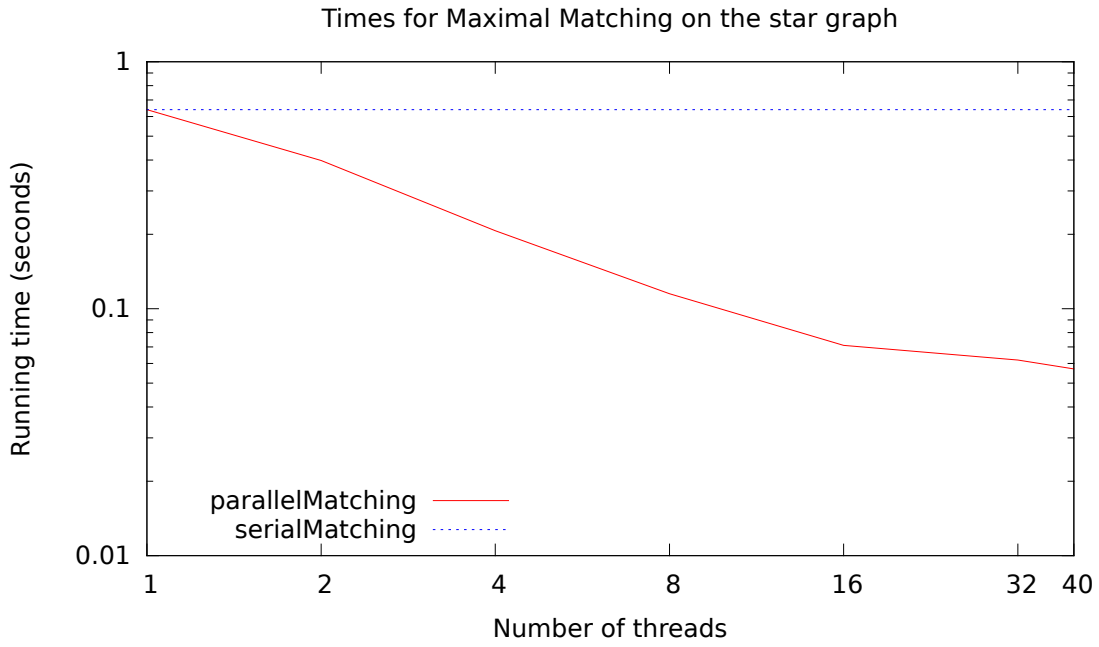


Figure 7: Maximal Matching times on the star graph (log-log scale)

due to the hardware optimizing writes to a single location (recall Figure 3 when $x = 1$). The exponential and comb graphs also cause high sharing due to the many incident edges on the high-degree vertices. The reservations are performed with write-with-min and Table 1(b) shows that this implementation performs well even on high-sharing graphs. Figure 7 shows that matching achieves good scalability on the high-sharing star graph.

For minimum spanning forest, the star and exponential graphs exhibit high sharing. We show the times on the various graphs in Table 1(c) and see that even for the high-sharing graphs the MSF implementation performs well (less than 3 times worse than the lower-sharing inputs). Figure 8 shows that MSF scales well even under the high-sharing star graph.

Our results for remove duplicates are similar, and are left to the appendix due to lack of space.

6 Conclusion

In this work we have compared the performance of several operations that are used when threads concurrently write to a small number of shared memory locations. Operations such as plain writes, compare-and-swap and fetch-and-add perform poorly under such high sharing, whereas priority update performs much better and close to the performance of reads. Using priority updates also has other benefits such as determinism, progress guarantees and correctness guarantees for certain algorithms. We show experiments for several applications which use priority update and demonstrate that even for high-sharing inputs, these applications are efficient and get good speedup. Given these results, we believe the priority update operation serves as a useful parallel primitive and good programming abstraction, and deserves further study.

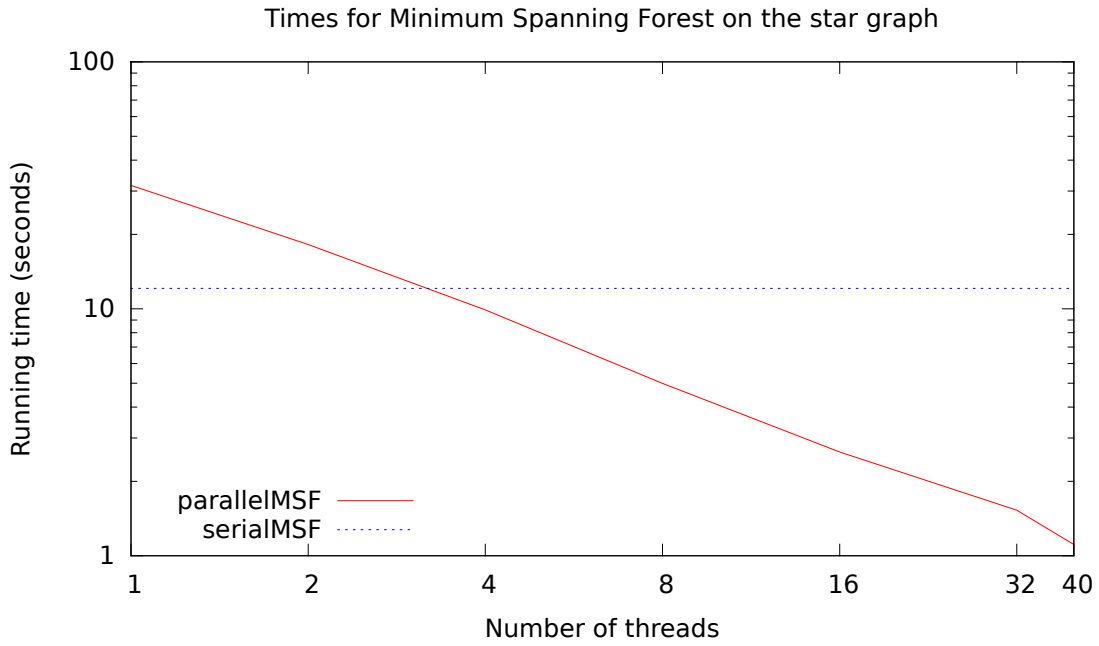


Figure 8: MSF times on the star graph (log-log scale)

(a) **Breadth-First Search**	3D-grid		random-local		rMat		comb		exponential		star	
	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)
serialBFS	2.28	–	2.98	–	3.39	–	3.96	–	1.19	–	0.35	–
writeMinBFS	4.03	0.325	6.21	0.319	7.54	0.314	6.41	0.24	3.09	0.21	0.757	0.064
writeBFS	4.03	0.314	5.39	0.239	6.86	0.296	9.9	3.1	2.9	0.253	0.759	0.063

(b) **Maximal Matching**	3D-grid		random-local		rMat		exponential		star	
	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)
serialMatching	0.38	–	0.783	–	1.0	–	0.677	–	0.823	–
writeMinMatching	1.1	0.083	1.78	0.12	2.79	0.138	1.23	0.086	0.64	0.068
writeMatching	1.02	0.068	1.74	0.089	2.44	0.129	1.13	0.084	0.594	0.101

(c) **Minimum Spanning Forest**	3D-grid		random-local		rMat		exponential		star	
	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)
serialMSF	4.31	–	7.28	–	9.53	–	7.56	–	12.1	–
parallelMSF	9.82	0.401	14.8	0.639	20.1	0.864	13.0	0.547	31.6	1.11

Table 2: Running times (seconds) of algorithms over various inputs. (40h) indicates the running time on 40 cores with hyper-threading and (1) indicates the running time on 1 thread.

References

[1] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. Internally deterministic algorithms can be fast. In *PPoPP*, 2012.

[2] Guy E. Blelloch, Jeremy T. Fineman, and Julian Shun. Greedy sequential maximal independent set and matching are parallel on average. In *SPAA*, 2012.

[3] Guy E. Blelloch, Phillip B. Gibbons, and Harsha Vardhan Simhadri. Combinable memory-block transactions. In *SPAA*, 2008.

[4] Guy E. Blelloch, Harsha Vardhan Simhadri, and Kanat Tangwongsan. Parallel and I/O efficient set covering algorithms. In *SPAA*, 2012.

[5] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-mat: A recursive model for graph mining. In *SDM*, 2004.

[6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.

[7] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 2008.

[8] Giovanni Della-Libera and Nir Shavit. Reactive diffracting trees. *J. Parallel Distrib. Comput.*, 2000.

[9] Zhen Fang, Lixin Zhang, John B. Carter, Ali Ibrahim, and Michael A. Parker. Active memory operations. In *Proc. 21st ACM Int'l Conf. on Supercomputing*, 2007.

[10] Panagiota Fatourou and Nikolaos D. Kallimanis. Revisiting the combining synchronization technique. In *PPoPP*, 2012.

[11] Alexandra Fedorova, Sergey Blagodurov, and Sergey Zhuravlev. Managing contention for shared resources on multicore processors. *Commun. ACM*, February 2010.

[12] Allan Gottlieb, R. Grishman, Clyde P. Kruskal, C. P. Mcauliffe, Larry Rudolph, and Mark Snir. The NYU Ultracomputer—designing an MIMD parallel computer. *IEEE Trans. Comput.*, 1983.

[13] Allan Gottlieb, Boris D. Lubachevsky, and Larry Rudolph. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM Trans. Program. Lang. Syst.*, 1983.

[14] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *SPAA*, 2010.

[15] J. Jaja. *Introduction to Parallel Algorithms*. Addison-Wesley Professional, 1992.

[16] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, February 1991.

[17] John M. Mellor-Crummey and Michael L. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. In *PPOPP*, 1991.

[18] John M. Mellor-Crummey and Michael L. Scott. Synchronization without contention. *SIGPLAN Not.*, April 1991.

[19] Mark S. Papamarcos and Janak H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *ISCA*, 1984.

[20] Larry Rudolph and Zary Segall. Dynamic decentralized cache schemes for mimd parallel processors. In *ISCA*, 1984.

[21] Nir Shavit and Asaph Zemach. Diffracting trees. *ACM Transactions on Computer Systems*, 1996.

[22] Nir Shavit and Asaph Zemach. Combining funnels: a dynamic approach to software combining. *J. Parallel Distrib. Comput.*, November 2000.

[23] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. Brief announcement: the problem based benchmark suite. In *SPAA*, 2012.

[24] Guy L. Steele Jr. Making asynchronous parallelism safe for the world. In *ACM POPL*, 1990.

[25] William E. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Trans. Computers*, 1988.

[26] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *ASPLOS*, 2010.

Remove Duplicates	allDiff		$\sqrt{n}$-unique		trigrams		allEqual	
Algorithm	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)
serialRemDups	3.42	–	0.365	–	0.973	–	0.257	–
writeMinRemDups	3.64	0.089	0.457	0.025	1.12	0.035	0.323	0.026
writeRemDups	3.59	0.087	0.484	0.03	1.1	0.297	0.405	4.35

Table 3: Remove Duplicates times (seconds). (40h) indicates the time on 40 cores with hyper-threading and (1) indicates the time on 1 thread.

A Appendix

A.1 Bounds for the Adversarial Model

Lemma 4 *The total cost for performing n priority updates (with random distinct values) to a single location using p cores under the adversarial model is $O(\frac{n}{p} + cp \ln n)$ with high probability.*

Proof. By Lemma 1, the number of random updates is $O(\ln n)$ with high probability. We now prove the number of attempts is at most $O(p \ln n)$, which implies the lemma. We say that a CAS fails due to the ith update if the old value conditioned on in the CAS is that of the $(i-1)$th update. Then there can be at most 1 CAS failure due to the ith update on each core, as any subsequent priority update on the same core would read the ith update and hence only fail due to a later update. There can thus be at most $p-1$ CAS failures in total for each update, for a total of $O(p \ln n)$ CAS attempts. □

In the adversarial model, the bound of Lemma 4 generalizes to $O(\frac{n}{p} + cmp \ln(\frac{n}{m}))$—for n operations the cost of reads is still $O(\frac{n}{p})$; now each location i can cost up to $O(cp \ln(n_i))$ leading to a total contribution of $O(\sum_{i=1}^{m} cp \ln(n_i))$ which is maximized when $n_i = \frac{n}{m}$ for all i.

Theorem 5 *The total cost for performing n priority updates (with random distinct values) to m randomly chosen locations under the adversarial model is $O(\frac{n}{p} + cmp \ln(\frac{n}{m})$ with high probability.*

A.2 Studying a False Sharing Anomaly

As seen in Figure 3(a), under false sharing write-with-min performs very poorly at 1024 locations. To hone in on this problem we studied the two machines under varying ratios of reads to writes to a small set of locations. Figure 9 shows the times for concurrently performing $10^8 - x$ reads and x writes on 1024 randomly chosen (adjacent) memory locations on each of the two architectures for varying x. Note that even for low fractions of writes (0.0001), the Intel Nehalem performs an order of magnitude worse than the AMD Operton architecture (which is a slower machine). This suggests that on the Intel Nehalem even when the vast majority of operations on a location are reads, there is still a big performance penalty from the cache coherence protocol of the very few writes. This effect is the cause of the hump in Figure 3(a) around the 1000 location point.

A.3 Priority Updates in a Hash-based Dictionary

Using priority updates to a single location it is possible to implement a dictionary that supports insertions of $(key, value)$-pairs such that the values of multiple insertions of the same key will be combined with a priority update. This can be thought of as a generalization of priority updates in

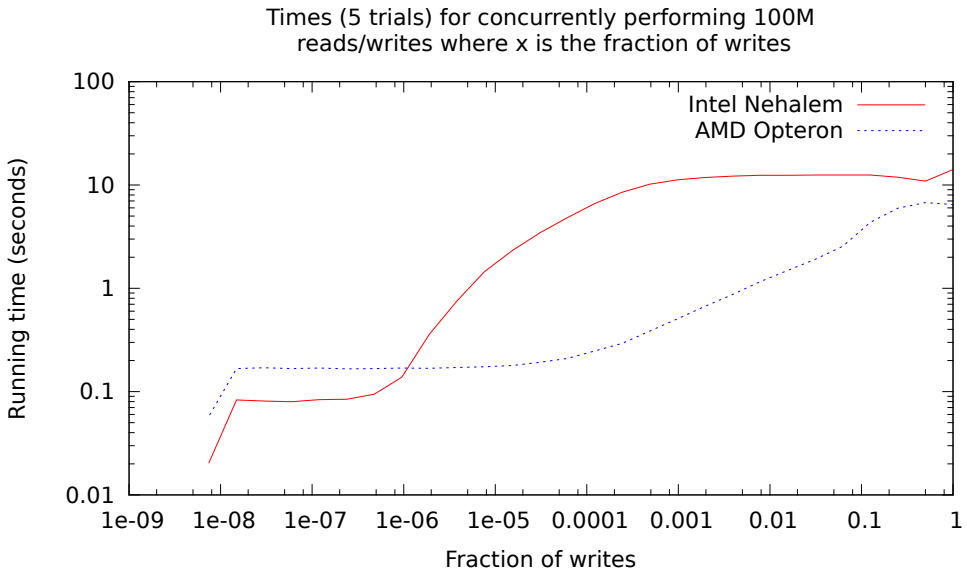


Figure 9: Times for performing 100 million reads/writes concurrently on 1024 random locations (with the fraction of writes varying) on the Intel Nehalem and AMD Opteron architectures (log-log scale).

which the "locations" are not memory addresses or positions in an array, but instead are indexed by arbitrary (hashable) keys. Applications of such key-based priority updates include making reservations on entries in a dictionary instead of locations in memory. Another application is to remove duplicates in a prioritized and/or deterministic way. For example we might have a large set of documents and want to keep only one copy of each word, but want it to be the first occurrence of the word (we assume the work is tagged with some other information that distinguishes occurrences, such as their location).

For this purpose we implemented a hash table-based dictionary that supports priority inserts. The table is based on linear probing and once a location already containing the same key or an empty location has been found, a priority update is used on the location with the priorities being based on the value associated with the key. We have found that this implementation is not much more expensive than a priority update directly on a location instead of on a key. The reason is that when using linear probing, there tends to be only one cache miss on the first location looked at, and if the location is full, the following locations are likely to be on the same cache line.

Experimental Study. The inputs to the Remove Duplicates problem is a sequence of $(key, value)$ pairs, and the return value is a sequence containing a subset of the input pairs such that no elements contain the same key. For pairs with equal keys, the pair that is kept is determined based on the $value$ of the keys. We use the sequence inputs from Table 4. The ***allDiff*** sequence contains pairs all with different keys. The $\sqrt{n}$***-unique*** sequence contains $\sqrt{n}$ copies of each of $\sqrt{n}$ unique keys. The ***allEqual*** sequence contains pairs with all the same key. Finally the ***trigrams*** sequence contains string keys drawn from a real-world text. The values of the pairs are random integers. The level of sharing at a location in the hash table is a function of the number of equal keys inserted at the location. Hence sequences with many equal keys will exhibit high sharing

Input	Size	Sharing Level
allDiff	10^7	Low
$\sqrt{n}$-unique	10^7	Medium
trigrams	10^7	Medium
allEqual	10^7	High

Table 4: Inputs for Remove Duplicates

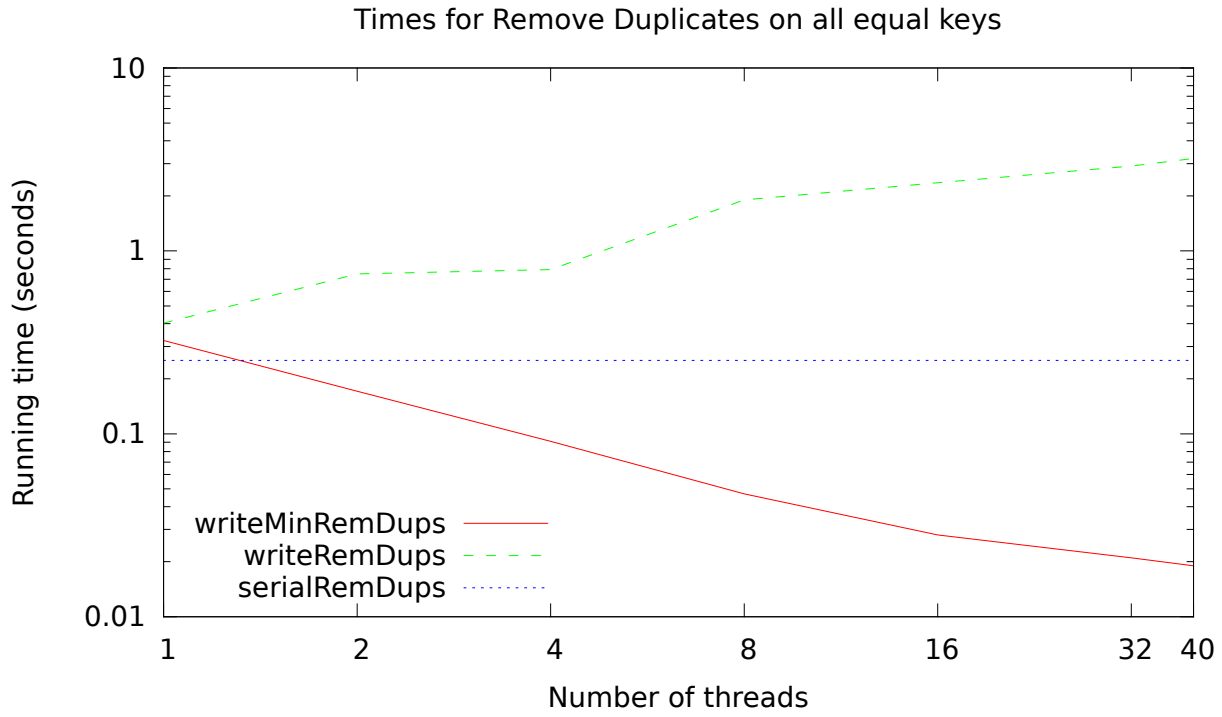


Figure 10: Remove Duplicates times on the allEqual sequence (log-log scale)

whereas sequences with few equal keys will have low sharing. We implement two versions of the hash table, one which arbitrarily chooses an element (***writeRemDups***) to keep when duplicate elements are inserted and the other which uses a write-with-min on the element's value to determine which element to keep (***writeMinRemDups***). In Figure 10, we compare the performance of writeRemDups and writeMinRemDups on the sequence of all equal keys, which exhibits the highest sharing. WriteMinRemDups scales gracefully with an increasing number of threads, while on a large number of threads, writeRemDups performs an order of magnitude worse than writeMinRemDups. Furthermore, for the high-sharing inputs, writeMinRemDups performs just as well as it does for lower-sharing inputs, as shown in Table 3.