Reducing Contention Through Priority Updates

Julian Shun
Carnegie Mellon University
jshun@cs.cmu.edu

Guy E. Blelloch
Carnegie Mellon University
guyb@cs.cmu.edu

Jeremy T. Fineman
Georgetown University
jfineman@cs.georgetown.edu

Phillip B. Gibbons
Intel Labs, Pittsburgh
phillip.b.gibbons@intel.com

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.

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 theoretically. 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 experiments showing that a variety of such algorithms and data structures perform well under high degrees of sharing. Given the results, we believe that 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.

Categories and Subject Descriptors: D.1.3 [Programming Techniques]: Concurrent Programming—*Parallel programming*

Keywords: Memory Contention, Parallel Programming

SPAA'13, June 23–25, 2013, Montréal, Québec, Canada.

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 because 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 of 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 desig-

nated 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.

Throughout the paper we will use the term *sharing* to indicate that a location is shared among many parallel operations, and *contention* to indicate a performance problem due to such sharing.

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—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 integers, where the minimum value written has priority. The priority update, however, can also be used when values do not fit in a hardware "word". For example the values could be character strings represented as pointers to the string stored in a memory word, or complex structures where a subfield is compared. The operation is therefore more general than what could be reasonably expected to be implemented in hardware.

We provide evidence that the priority update operation serves as a good abstraction for programmers of shared memory machines because it is 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 3.1.) 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 priority update (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 only the early operations will attempt to set a location, and the rest will access the already set location in shared mode.

The priority update can be implemented in software with a read, a local comparison, and a compare-and-swap. The compare-and-swap is needed only when the value being written is smaller than the existing value. Thus, when applied

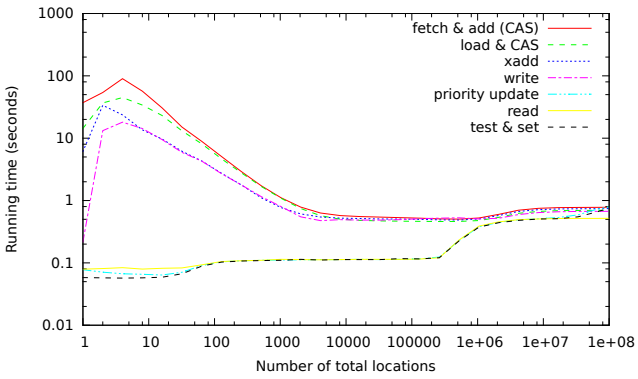


Figure 1. Impact of sharing on a variety of operations. Times are for 5 runs of 100 million operations to varying number of memory locations on a 40-core Intel Nehalem (log-log scale). Since the number of operations is fixed, fewer locations implies more operations sharing those locations.

with random values (or in a random order) most invocations of priority update only *read* shared data, which is why the running time nearly matches the read curve, and is effectively the same as the test-and-set curve. The curve shows that the high sharing case is actually the best case for a priority update. This implies the user need not worry about contention, although, as with reads, the user might still need to worry about the memory footprint and whether it fits in cache—the steps in the curve arise each time the number of locations no longer fits within a cache at a particular level.

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.

Priority updates are 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 (earlier iterates have higher priority). 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 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 single-source 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 "locations" are based on keys. Each insert consists of a key-value pair, and updates the data associated with the key if either the key does not appear in the dictionary or the new value has higher priority.

In this paper we describe these algorithms and present experimental results. We study the performance of several of these algorithms including BFS, Kruskal's minimum spanning forest algorithm, a maximal matching algorithm and a dictionary-based remove duplicates algorithm. We present timing results for inputs with high sharing, and for BFS and remove duplicates, 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 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 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 several examples of algorithms for a number of important problems that demonstrate a variety of ways to benefit from priority updates. 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, and providing a comparison to implementations using alternative primitives.

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., the value ending up in the location will be the same independent of the ordering of the updates.

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 more complex structures). 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.

```
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.

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 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.

Although the version of priority update we described does not return a value, it is easy to extend it to return the old value stored in the location. Indeed in one of our applications we can make use of this feature.

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 multicores) 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 because 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 much more expensive than simply having one core do all the writes. On a system with just 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 demonstrate.

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. Our experiments show that 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 experimental evidence (Section 3.1) and theoretical justification (Section 3.2) of its efficiency.

3.1 Experimental Measurements of Contention

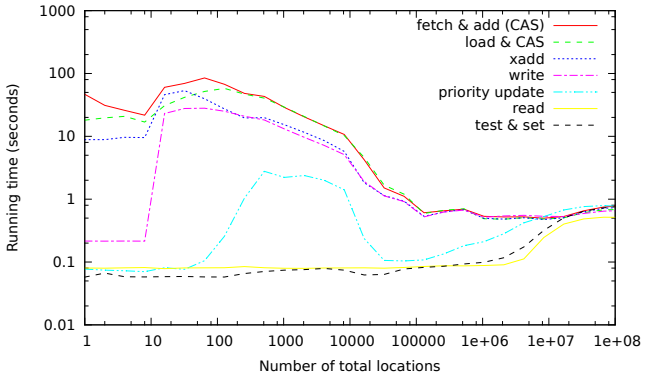
We study the cost of contention under varying degrees of sharing on two contemporary shared memory multicores (from Intel and AMD) for a variety of memory operations—priority update (using write-with-min), test-and-set, fetch-and-add using CAS, fetch-and-add using the x86 assembly instruction `xadd`, load-and-CAS, (plain) write, and read[1]. We compare the performance of priority update (write-with-min) when values are random versus when values arrive in a decreasing order (the worst case). We also study the performance of priority update where the comparison is on character strings.

The Intel machine that we use is a 40-core (with hyper-threading) Nehalem-EX machine with 4×2.4GHz Intel 10-core E7-8870 Xeon processors and 256GB of main memory. The AMD machine that we use is a 64-core machine with $4\times$ 2.4GHz AMD 16-core Opteron 6278 processors and 188GB of main memory. 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.

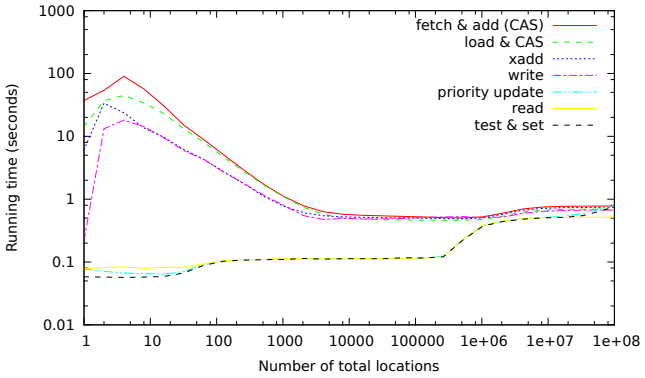
In the experiments, we perform 10^8 operations on a varying number of random locations. On each machine we performed two sets of experiments. The first set of experiments choose the locations randomly in $[0, x)$ where x is the total number of locations written to and locations 0 through x appear contiguously 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 an integer 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, priority update outperforms plain write, both versions of fetch-and-add, and load-and-CAS by orders of magnitude. Due to an Intel anomaly (see the Appendix), there is a spike in the running time for priority update between 256 and 8192 locations, but even with this anomaly, priority update 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 priority update is very close to the performance of both test-and-set and read. For writing to 10^8 locations (the lowest degree of sharing), priority update 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 priority update and write does more work than test-and-set). We conjecture this behavior to be due to the branch in both priority update 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, because 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).

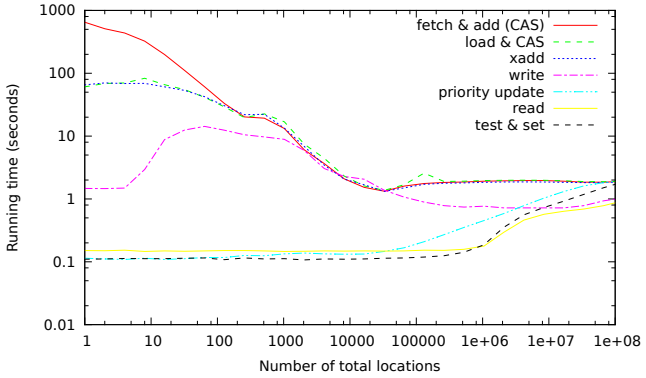
[1]The read includes a write to local memory to get around compiler optimizations.



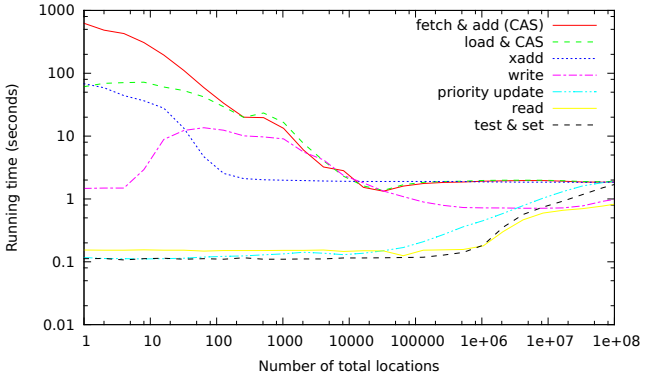
(a) High false sharing on 40-core Intel machine



(b) Low false sharing on 40-core Intel machine



(c) High false sharing on 64-core AMD machine



(d) Low false sharing on 64-core AMD machine

Figure 3. Impact of sharing. Times are for 5 runs of 100 million operations to varying number of memory locations on Intel and AMD machines. under 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.

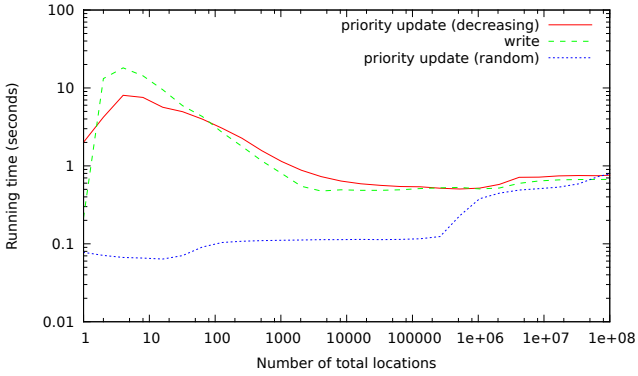


Figure 4. Comparing priority update (write-with-min) on random values vs. decreasing values. Times are for 5 runs of 100 million operations to varying number of memory locations with low false sharing on the 40-core Intel machine (log-log scale).

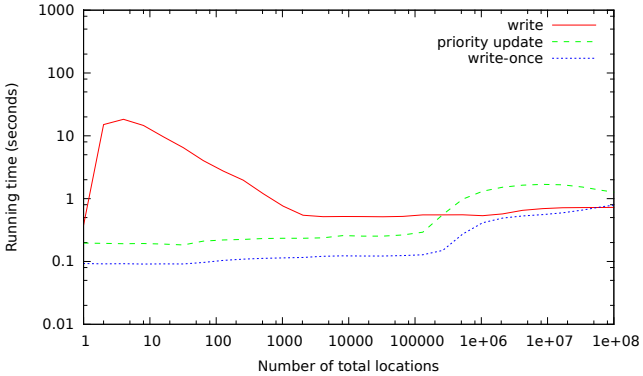


Figure 5. Priority update on character strings based on trigram distribution of the English language. Times are for 5 runs of 100 million operations to varying number of memory locations with low false sharing on the 40-core Intel machine (log-log scale).

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 priority update operation observed for the Intel machine does not appear for the AMD machine. Except for this anomaly, the performance on the Intel machine is better than the performance on the AMD machine.

Note that for priority update, the relative order of values over time greatly impacts the number of update attempts and hence the cost. In the above experiments, the priority update uses random values, which is also the setting studied in our theoretical analysis. The worst case is when the values have increasing priorities over time, as this incurs the most update attempts. With write-with-min, for example, this case arises when values occur in decreasing order. Figure 4 shows that the performance of this case (labeled "priority update (decreasing)") is much worse than the random case.

Figure 5 shows the performance of priority update, where the comparison is on character strings based on the trigram distribution of the English language (the *trigrams* input in Section 5). This uses the more general form of priority update as the comparison function requires dereferencing the pointers to the strings. We also compare the performance to using plain writes and write-once to update the values at the shared locations. Note that no pointer dereferencing needs to be done in these versions—plain write just overwrites the pointer at the location, and write-once writes the pointer to the location only if it is empty. Similar to the performance on integer values shown in Figure 3, the performance of the version using plain writes is an order of magnitude worse than the priority update and write-once versions. The write-once version is faster than the priority update version, and the gap is more significant here (compared to priority update vs. test-and-set in Figure 3) due to the cost of pointer dereferencing in the priority update.

3.2 Priority Update Performance Guarantees

As discussed in Section 2, priority update 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 our analysis shows that the number of updates is small, with most invocations only reading the shared data. We begin with a straightforward analysis of sequential updates and then extend the analysis to a collection of parallel priority updates. There are two main challenges in the parallel analysis: developing a cost model that reasonably captures the read/write asymmetry in the coherence protocol, and coping with the fact that different access delays cause operations to fall out of sync.

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 $E[X_1 + \cdots + X_\phi] = E[X_1] + \cdots + E[X_\phi] = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{\phi} = H_\phi$. Applying a Chernoff bound gives 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 also assume that the comparison function $>_p$ takes constant time, although our analysis can be easily extended to non-constant time comparison functions.

We assume that a collection of n priority updates are ordered[2] and have values corresponding to a random permutation of the set $\{1, \ldots, n\}$, with 1 being the highest priority and each location initialized to a special lowest-priority value ∞. This is equivalent to randomly ordering the priority updates and then assigning each value to its relative rank in the total order. While we assume the values are distinct, the bounds can be readily sharpened to take into account the actual number of occurring priorities, as in Lemma 1. Note that the actual values of the priority updates do not matter, as long as the order of the priority updates is randomized.

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.[3] 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 time 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 are 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, c time 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 arbitrarily order any outstanding CAS and read operations (e.g., based on the locations being written), but without considering the values being written.

3.2.1 Bounds for the Fair Model

To analyze priority updates to a single location in the fair model, we view operations as being processed in rounds induced by the queue ordering. Each round processes p operations, one per core, which may be either of the two steps of a priority update: a read or a CAS.[4] 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 to v_j, and commits to either performing a CAS in round $j+1$ or skipping the CAS attempt step and proceeding to the

[2]Cores have disjoint subsequences of this ordering, determined at runtime by the scheduler.

[3]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 while the line could be invalidated in between those two steps, our experiments on both Intel Nehalem and AMD Opteron multicores support assuming it is not.

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

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 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 $\{1, \ldots, n\}$.

LEMMA 2. *The expected total time for performing n randomly ordered priority updates to a single location using p cores under the fair model is $O(\frac{n}{p} + c \ln n + cp)$.*

PROOF. By Lemma 1, there are $O(\ln n)$ successful updates, so the goal is to bound the number of unsuccessful CAS attempts. We start by bounding the number of priority updates that include at least one 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 $\frac{n}{2^i}$ (recall that we are assuming values are the relative ranks), and b) a successful CAS occurs. We want 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 because 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 (taking c time) can be charged to the preceding CAS attempt, only doubling the time. The first read takes c time, and the remaining reads and all local computation take $O(\frac{n}{p})$ time, completing the proof. □

The above results are for performing priority updates to a single location. Now we analyze the time 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 individual 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 expected total time for performing n randomly ordered priority updates to m randomly chosen locations under the fair model is $O(\frac{n}{p} + cm \ln(\frac{n}{m}) + (cp)^2)$.*

PROOF. According to the analysis of Lemma 2, 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 $O(\frac{n}{p} + cm \ln(\frac{n}{m}) + cpm)$ follows by maximizing the logarithmic term (setting $n_i = \frac{n}{m}$ for all i) and multiplying by m locations. This bound is pessimistic, so we will improve it for $m > p$. The $O(cm \ln(\frac{n}{m}))$ term seems inherent because each update invalidates the line in all other caches, so the time to reload those lines later is $O(cpm \ln(\frac{n}{m}))$ (which is divided across p cores). Our goal is to reduce the $O(cpm)$ 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 this location during a round containing a successful CAS. The maximum duration 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 per round is $O(\frac{cp^2}{m})$, only some of which may actually perform a CAS attempt. As in Lemma 2, the likelihood of performing a CAS attempt decreases geometrically per 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 taking c time. □

3.2.2 Bounds for the Adversarial Model

We now analyze priority updates under the adversarial model. Recall that in the adversarial model, an adversary may order any outstanding CAS and read operations arbitrarily (e.g., based on the locations being written), but without considering the actual values being written.

LEMMA 4. *The total time for performing n randomly ordered priority updates 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. 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 per update, for a total of $O(p \ln n)$ CAS attempts. □

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

THEOREM 5. *The total time for performing n randomly ordered priority updates to m randomly chosen locations under the adversarial model is $O(\frac{n}{p} + cpm \ln(\frac{n}{m}))$ with high probability.*

For reasonably sized n, the bounds in this section (under both models) 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.

4. APPLICATIONS OF PRIORITY UPDATE

Priority updates are well-suited to a widely applicable two-phase programming style, which we call *update-and-read* in its general form, and *reserve-and-commit* in a special case. An ***update-and-read*** program alternates two types of 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 that it would read from or write to. In the *commit* (read) phase, each object in parallel checks whether it holds a reservation on its neighborhood, and if so, performs the desired operations. 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, write-once or plain write. The priority update 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 the 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 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 speed-ups (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 exception is connected components, where a priority update 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 single-source 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 r 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 modifications of the BFS implementations from the publicly available problem-based benchmark suite (PBBS) [23]. 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 frontier vertex checks to see if successfully reserved its neighbor, and if so it adds the neighbor to the next frontier. Since only one frontier vertex will successfully reserve a neighbor, there will be no duplicates on the next frontier.

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

We also use a version of deterministic BFS that has only one phase per round and returns the same BFS tree as a sequential implementation. This version uses a priority update on pairs $(index, parent)$, where *index* is a vertex's parent's order in a sequential BFS traversal, and *parent* is the vertex's parent's ID. The priority update does a min-comparison only on the *index* field of the pair. All frontier vertices perform priority updates to neighbors and if it successfully updates the neighbor's location, it adds the neighbor to the next frontier in the same phase. Since this implementation only has a single phase, it allows for duplicate vertices on the frontier (multiple priority updates may succeed on the same neighbor). The form of priority update used here is more general than write-with-min.

Maximal Matching. The ***maximal matching*** (MM) problem takes an undirected graph $G = (V, E)$ and returns a

subset $E' \subseteq E$ such that no two edges in E' have an endpoint in common (matching) and all edges in $E \setminus E'$ share at least one endpoint with an edge in E' (maximal). The reserve-and-commit style can be used to solve the MM problem [2]. During the reserve phase, each edge checks if either of its endpoints have been matched; if so the edge removes itself from the graph and otherwise it reserves both endpoints by using a priority update (write-with-min) with the edge's unique ID. During the commit phase, every remaining edge checks if both of its endpoints contain its reservation; if so, it joins the matching and removes itself from the graph. The algorithm can also be implemented using write-once or plain writes, but forward progress is not guaranteed because it is possible that no edge succeeds in reserving both of its endpoints in an iteration.

Connected Components. For an undirected 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 $C_1 \cup \cdots \cup C_k = V$. A simple vertex-based algorithm assigns each vertex a unique ID 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 priority update (write-with-min) to all of its neighbors' IDs. This is an example of using priority update 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 with 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) [6]. Kruskal's algorithm considers edges in sorted order by weight and iteratively adds edges that connect two different 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, a reserve-and-commit style is applicable: During the reserve phase each edge that has endpoints in separate components writes its weight/ID to the component containing each endpoint, and during the commit phase each edge checks whether its ID was written to at least one component—if so, it joins the MSF. As with connected components, the priority update (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].

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 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).

Our experiments use modifications of the hash table-based dictionary from PBBS [23] that supports priority inserts. The table is based on linear probing and once a location 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.

Input	Num. Vertices	Num. Directed Edges	Sharing Level
3D-grid	10^7	6×10^7	Low
random-local	10^7	10^8	Low
rMat	2^{24}	10^8	Medium
4-comb	2.5×10^7	10^8	High
exponential	5×10^6	1.1×10^8	High
4-star	5×10^7	10^8	High

Table 1. Inputs for graph applications.

Other Applications. Priority updates are applicable to other problems whose solutions are implemented using deterministic reservations [1]. These problems include Delaunay triangulation and refinement [7], maximal independent set and randomly permuting an array. In most of these cases (as with maximal matching), write-once and plain write 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 the random numbers are fixed. A priority update (write-with-min) can be naturally applied to a single-source shortest paths implementation to asynchronously update potentially shorter paths to vertices. A write-once or plain write implementation would not be correct here, since we must store the shortest path to each vertex. Priority updates are also 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 3.1. For sequential programs, we used the `g++` 4.4.1 compiler with the `-O2` flag. For the breadth-first search, maximal matching, minimum spanning forest, and remove duplicates applications, we ran experiments on inputs that exhibit varying degrees of sharing. 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

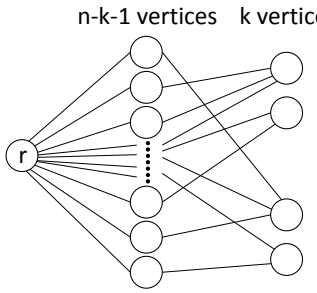


Figure 6. k-comb graph (used for BFS experiments).

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 undirected 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 ***k-comb*** graph is a three layered graph (see Figure 6) 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 operations 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 ***4-star*** graph is a graph with four "center" vertices and each of the $n - 4$ remaining vertices is connected to a randomly chosen center vertex (total of $2(n - 4)$ directed edges).

In BFS, because many vertices may compete to become the parent of the same neighbor, there can be high sharing. The k-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 6). We chose to model sharing on k-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 four versions of parallel BFS which deal with reserving neighbors and placing them onto the frontier differently. The first version uses a priority update with the minimum function (***priorityUpdate-BFS***) in a two-phase update-and-read style; the second uses a priority update in a single phase, produces the sequential BFS tree but allows for duplicate vertices on the frontier (***seqOrder-BFS***); the third uses a test-and-set (***testSet-BFS***); and the fourth uses a plain write (***write-BFS***) (see Section 4 for details). Figure 7 compares the four BFS implementations and the sequential BFS implementation (***serial-BFS***) as a function of number of cores on the 4-comb graph. Table 2(a) shows the running times for each of the BFS implementations on all of our graphs. The (nondeterministic) test-and-set implementation is the fastest because only one actual write is done per vertex. However the priority update implementations do not do much worse even on the high-sharing comb graph while the plain-write implementation does poorly on it (even worse than serial-BFS). The two-phase and one-phase priority update implementations are comparable in performance. Using a family of k-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 8 shows that for values of k up to around 10000, priorityUpdate-BFS and seqOrder-BFS outperform write-BFS, by nearly an order of magnitude for small k, and is almost as fast as testSet-BFS. For higher values of k where there is little sharing, priorityUpdate-BFS and seqOrder-BFS are slower than writeBFS due to the overhead of the test and compare-and-swap, however they are deterministic. For values of k less than 2000 (high sharing), write-BFS is worse than even the sequential implementation.

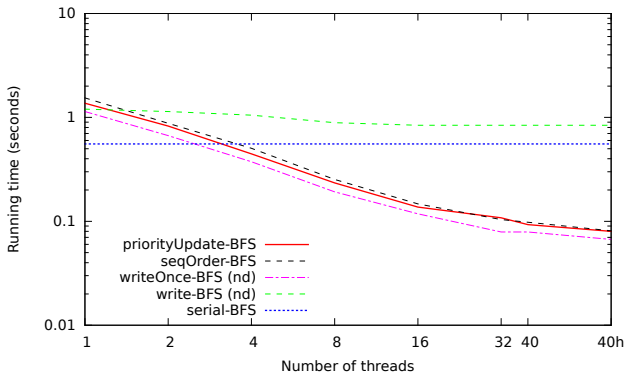


Figure 7. BFS times vs. number of cores on the 4-comb graph (log-log scale). (nd) indicates a non-deterministic implementation.

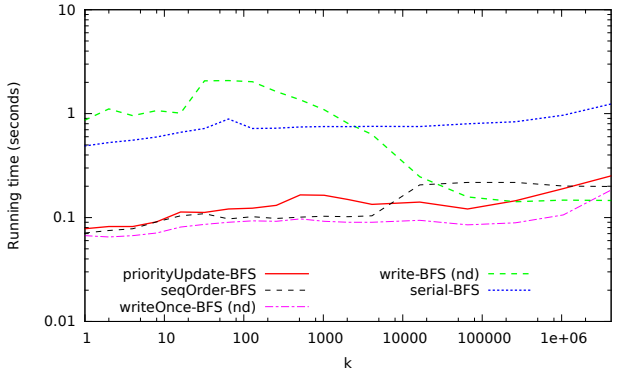


Figure 8. BFS times on different k-comb graphs with $n = 2.5 \times 10^7$ on 40 cores (log-log scale). Lower k means higher sharing. (nd) indicates a non-deterministic implementation.

For maximal matching and minimum spanning forest, the 4-star and exponential graphs exhibit high sharing. We show the times for implementations using priority updates and also serial implementations on the various graphs in Tables 2(b) and 2(c). We see that even for the high-sharing graphs the implementations performs well (less than 3 times worse than the lower-sharing inputs on 80 hyper-threads).

(a)

Breadth-First Search	3D-grid		random-local		rMat		4-comb		exponential		4-star	
	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)
serial-BFS	2.03	–	2.77	–	3.13	–	0.555	–	1.19	–	0.317	–
priorityUpdate-BFS	4.03	0.307	7.02	0.247	8.37	0.306	1.38	0.08	3.18	0.199	0.885	0.066
seqOrder-BFS	3.12	0.339	5.42	0.258	6.28	0.365	1.54	0.081	3.05	0.285	0.849	0.064
testSet-BFS (nd)	2.66	0.25	4.8	0.16	5.45	0.211	1.14	0.066	2.17	0.097	0.664	0.055
write-BFS (nd)	4.3	0.28	6.13	0.246	7.74	0.298	1.2	0.954	3.18	0.224	0.888	0.063

(b)

Maximal Matching	3D-grid		random-local		rMat		exponential		4-star	
	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)
serial-Matching	0.527	–	0.764	–	1.0	–	0.674	–	0.823	–
priorityUpdate-Matching	1.41	0.091	1.8	0.113	2.82	0.142	1.27	0.082	0.641	0.062

(c)

Minimum Spanning Forest	3D-grid		random-local		rMat		exponential		4-star	
	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)
serial-MSF	5.3	–	7.29	–	9.54	–	7.45	–	13.3	–
priorityUpdate-MSF	10.7	0.455	14.1	0.614	19.0	0.816	12.2	0.53	29.4	1.04

(d)

Remove Duplicates Algorithm	allDiff		$\sqrt{n}$-unique		trigrams		allEqual	
	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)
serial-RemDups	3.25	–	0.364	–	0.975	–	0.255	–
priority-UpdateRemDups	3.31	0.078	0.442	0.021	1.07	0.033	0.318	0.02
writeOnce-RemDups (nd)	2.16	0.072	0.433	0.021	1.03	0.035	0.312	0.021
write-RemDups (nd)	3.3	0.083	0.471	0.028	1.05	0.291	0.386	3.19

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. (nd) indicates a non-deterministic implementation.

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

Table 3. Inputs for Remove Duplicates.

The input 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 that contains only one element of any given *key* from the input. We use the hash-based dictionary described in Section 4 to solve this problem. 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 3. 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 based on the trigram distribution of the English language. 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, whereas sequences with few equal keys will have low sharing. We show experiments for three versions of the parallel hash table which deal with insertions of duplicate keys differently. The first version, ***write-RemDups***, always performs a write of the value to the location when encountering a key that has already been inserted; the second version, ***writeOnce-RemDups***, does not do anything when encountering an already inserted key; and the last version, ***priorityUpdate-RemDups***, uses a priority update with the minimum function on the values associated with the keys when encountering duplicate keys.

In Figure 9, we compare the performance of the various parallel implementations, along with a serial implementation (***serial-RemDups***) on the sequence of all equal keys, which exhibits the highest sharing. The priority update and

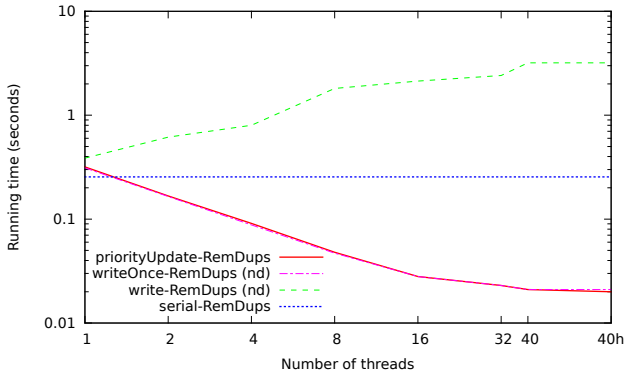


Figure 9. Remove Duplicates times on the allEqual sequence (log-log scale). (nd) indicates a non-deterministic implementation.

write-once implementations scale gracefully with an increasing number of threads, while on a large number of threads, the plain write implementation performs an order of magnitude worse. The priority update and write-once implementations of remove duplicates have similar performance, but the former also has the advantage that it is deterministic. The timings for all of the inputs are shown in Table 2(d).

6. CONCLUSION

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 that 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.

Acknowledgments. 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.

References

[1] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and J. Shun. Internally deterministic algorithms can be fast. In *PPoPP*, 2012.

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

[3] G. E. Blelloch, P. B. Gibbons, and H. V. Simhadri. Combinable memory-block transactions. In *SPAA*, 2008.

[4] G. E. Blelloch, H. V. Simhadri, and K. Tangwongsan. Parallel and I/O efficient set covering algorithms. In *SPAA*, 2012.

[5] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-mat: A recursive model for graph mining. In *SDM*, 2004.

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

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

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

[9] Z. Fang, L. Zhang, J. B. Carter, A. Ibrahim, and M. A. Parker. Active memory operations. In *SC*, 2007.

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

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

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

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

[14] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *SPAA*, 2010.

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

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

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

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

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

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

[21] N. Shavit and A. Zemach. Diffracting trees. *ACM Trans. Comput. Syst.*, 1996.

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

[23] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan. Brief announcement: the Problem Based Benchmark Suite. In *SPAA*, 2012.

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

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

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

APPENDIX

Studying a False Sharing Anomaly. As seen in Figure 3(a), under false sharing priority update (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 10 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 Opteron (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.

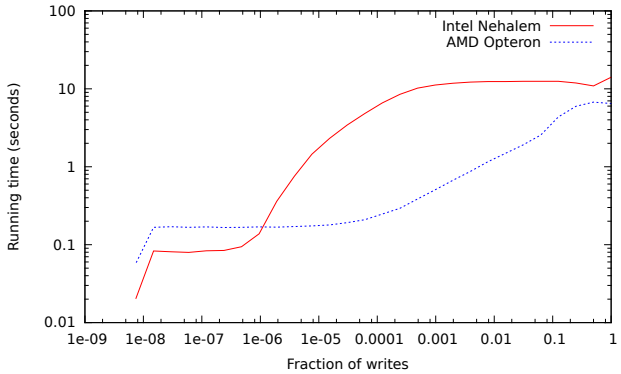


Figure 10. Studying a false sharing anomaly. Times are for 5 runs of 100 million concurrent reads/writes to 1024 random locations (with the fraction of writes varying) on the 40-core Intel Nehalem and the 64-core AMD Opteron machines (log-log scale).