

# An Investigation of the Cost of Agreement in Synchronization

Philip W. Howard

Portland State University  
pwh@cecs.pdx.edu

**Abstract.** The prevalence of multicore computers is leading to an increase in the amount of concurrent software. Unfortunately, parallelization does not necessarily improve performance. A concurrent program may run slower than the sequential equivalent. It may even scale negatively—performance may get worse as more processors are added.

It is important to understand the causes of this poor performance because the trend to multicore is likely to continue. This paper presents a suite of performance benchmarks which are used to examine the cost of synchronizing access to shared data. The components of the cost include the cost of blocking, the cost of atomic instructions, and the cost of sharing data. These costs can be described in terms of the need to agree on something. In the case of blocking, it is the need to agree on whose turn it is. In the case of atomic instructions, it is the need to agree on which CPU gets to execute the instruction next. In the case of sharing data, it is the need to agree on what the current value of a shared variable is. The benchmarks show that the agreement on the value of shared data is the main impediment to scalability.

The benchmark results are used to predict the scalability properties of several common synchronization approaches. The approaches include locking, non-blocking synchronization, and relativistic programming. Each approach's need-to-agree is shown to be a useful predictor for the approach's performance and scalability.

## 1 Introduction

When working on a group project, people sometimes spend more time coordinating schedules than they do performing productive work. The same can be true with processors in a multiprocessor system—the processors need to coordinate who does what when. This coordination is called synchronization. Without synchronization, there is the possibility of data corruption or program failure. With synchronization, there is the possibility of poor performance.

Synchronization performance has become particularly important in light of the move toward ever increasing parallelism in computer hardware. The next two sections examine hardware issues that impact synchronization and scalability.

### 1.1 The Move to Multicore

Up until 2003, processor clock rates doubled about every two years. During this time, an application’s performance could be improved in two ways: the development team could optimize the performance, or they could simply wait for the next generation processor to come out. The “wait” strategy was viable (assuming time to market constraints would tolerate the delay) because most applications would benefit from the increased performance of the next generation processors.

Since 2003, processor clock rates have held steady at about 3Ghz. To satisfy the desire for more processing power, manufacturers have been adding processing cores to their chips rather than increasing the clock rate. There is every indication that this trend will continue [1]. This change means that the “wait” strategy will not necessarily work anymore. The next generation of processors will not benefit an application unless that application *scales*—its performance must increase as more processors are added.

The trend to multicore processors means that scalability should be included as part of the correctness criteria of any synchronization technique used in building concurrent applications. If a synchronization technique prevents an application from scaling, the application can not take advantage of additional processors. If an application does not scale, there is no performance reason to make it concurrent. It might be better to eliminate the complexity of concurrency and keep the application sequential.

### 1.2 The UMA Myth

In a Uniform Memory Access (UMA) architecture computer there is a uniform cost for accessing any memory location. In a Non-Uniform Memory Access (NUMA) architecture computer the cost of accessing memory is different based on which memory location is accessed. Typically NUMA computers have some memory local to each processor. When a processor accesses its local memory, the access is fast. When a processor accesses memory that is local to another processor, the access is slower because of the communication involved.

Most developers think of modern multicore computers as UMA computers. Conceptually there is a single memory which all processing cores have direct access to. In reality however, there are multiple memories. Each processing core typically has its own cache memory. A processor can access data in its local cache much more quickly than it can access main memory or another processor’s cache.

The cache coherence mechanism attempts to preserve a coherent view of memory between processors, but there is a cost associated with this. The cache coherence mechanism causes values to move between caches and main memory so that all readers will see the most recently written value<sup>1</sup>. The cache coherence mechanism is critical for getting correct behavior out of programs running

---

<sup>1</sup> More precisely, the cache coherence mechanism will cause readers to see a value that is consistent with the computer’s memory consistency model.

on multiprocessor machines. However, the communication required to keep the caches coherent can have negative effects on the scalability of a program.

In order to write efficient code for a NUMA computer, the developer needs to be aware of where code and data resides. The best performance will be achieved if the code and data are kept in a local memory. In order to write scalable concurrent programs on multicore computers, programmers need to be aware of the cache mechanism in the same way that programmers of traditional NUMA machines need to be aware of memory layout.

### 1.3 The Problem Under Consideration

To build scalable synchronization techniques, we must understand what limits scalability. The hypothesis for this paper is that the scalability of synchronization techniques is determined by the extent to which they need to agree on something. The need-to-agree comes in several forms. The cache coherence mechanism allows all processors to agree on the value at a particular memory address. This agreement requires communication between the caches and this communication can limit scalability. This form of need-to-agree will be called memory hierarchy need-to-agree.

Another form of need-to-agree is algorithmic need-to-agree. An example of this is mutual exclusion. With mutual exclusion, only one thread at a time is allowed to access shared data. Mutual exclusion requires all threads to agree on whose turn it is. There has to be some algorithm that is used to come to this agreement.

A third form of need-to-agree is instruction level need-to-agree. Most modern processors have atomic instructions which guarantee that the entire instruction happens as a single operation. Interleaved operations from other processors are not allowed to affect the outcome of the atomic instruction. Atomic instructions need to agree on who gets to go first and what other operations are allowed to proceed in parallel. This is similar to algorithmic need-to-agree but it is enforced by the underlying hardware rather than by a software algorithm.

Each of these need-to-agree components contribute to the cost of a synchronization mechanism. This paper presents a series of benchmarks that isolate the cost of each of these need-to-agree components. The results of the benchmarks are then used to predict the performance and scalability of linked list operations using different synchronization mechanisms. Another set of benchmarks is presented that validate the predictions. Finally, conclusions are drawn that suggest future directions for research in this area.

## 2 Experimental Design

The need-to-agree costs presented in this paper were determined experimentally. Benchmarks were performed on a 16 processor system that allowed the benchmarks to show the need-to-agree impact on scalability for up to 16 processors.

Each benchmark measures the number of operations per second that can be performed with a given algorithm on a given number of processors. To accomplish this, a master thread starts  $N$  worker threads. The master thread synchronizes the worker threads so they start and stop collecting data at the same time. Once the master thread indicates that the worker threads are to start, they execute the algorithm in a loop until the master thread indicates it is time to stop. The number of times through the loop is the number of operations the worker thread performed.

Care was taken to minimize contention between worker threads. Counters were local to a thread and were cache line aligned so there would be no false sharing between threads. There were never more threads than processors so each thread could have a dedicated processor.

Each benchmark configuration was run sixteen times. The values reported in later sections are the average of the sixteen runs. Error bars are included in the charts which show the 90% confidence interval on the values.

The tests were performed on a system with four quad-core Intel Xeon E7310 processors (total of 16 processing cores). The processors were running at 1.6 Ghz. Each processing core had a private 32KB L1 instruction cache and a 32KB L1 data cache. Each pair of processing cores shared a 2MB L2 cache. The cache line size was 64 bytes. The system was running Linux kernel version 2.6.27. The code was compiled with gcc version 4.3.2 with optimization was set to `-O1`<sup>2</sup>.

### 3 The Need-to-agree with *Spinlocks*

This section presents an analysis of a *spinlock* implementation in terms of its need-to-agree. This analysis leads to a series of benchmarks that are used to isolate the component costs of the agreement.

The *spinlock* used in this paper is built on the Compare And Swap operation described in the next section. Later sections show an analysis of the need-to-agree in *spinlock* and describe variants which isolate the different components of the need-to-agree. It must be noted that of all the variants, only *spinlock* is a true lock. The other variants have a reduced need-to-agree; and because of this, the other variants do not guarantee mutual exclusion. The purpose of the other variants is to perform systematic testing of the cost of agreement.

#### 3.1 Compare And Swap

The Compare And Swap<sup>3</sup> (CAS) operation compares a memory location with an expected value. If the value at the memory location is the same as the expected

<sup>2</sup> Optimization level `-O1` inlined functions that were used for performing some low level operations (see documentation for `libatomicops`). Higher levels of optimization seemed to ignore important attributes such as `volatile`.

<sup>3</sup> A better name would be Compare And Update because the values are not actually swapped. Compare And Swap is the traditional name for this operation, so that name is used here.

value, then a desired value is stored at the memory location. Figure 1 gives pseudocode for the compare and swap operation. The return value can be used

```

boolean CAS(int *memory, int expected, int desired) {
    if (*memory == expected) {
        *memory = desired;
        return TRUE;
    } else {
        return FALSE;
    }
}

```

**Fig. 1.** Pseudocode for Compare And Swap operation

to determine if the swap took place or not.

Note that CAS is a need-to-agree operation because all processors must agree on who gets to perform the swap. If two processors performed the compare before either performed the swap, one of the swaps might get lost and never be seen. If the entire operation is performed atomically (as an uninterruptible unit), the compare of one of the processors would see the results of the other's swap.

Fortunately, most modern processors provide an atomic CAS (or equivalent) instruction. In the Intel x86 architecture the instruction is called Compare And Exchange (`cmpxchg`). The `cmpxchg` instruction can be made atomic by using the lock prefix. The lock prefix guarantees that if two processors attempt to perform a `cmpxchg` at the same time with the same parameters, only one will perform the swap [2].

### 3.2 Analysis of *Spinlock*

Given the CAS operation defined in the previous section, a *spinlock* can be created by placing the CAS in a while loop. Pseudocode for a *spinlock* is given in Fig. 2. The *spinlock* waits until the compare sees that the lock is UNLOCKED.

```

boolean spinlock(int *lock) {
    while (!CAS(lock, UNLOCKED, LOCKED))
        {}
}

boolean spin_unlock(int *lock) {
    *lock = UNLOCKED;
}

```

**Fig. 2.** Pseudocode for *spinlock* and *spin unlock*

It then sets the value to `LOCKED`.

Pseudocode for the unlock operation is also shown in Fig. 2. Note that there is no need-to-agree in the unlock because by convention, once a processor owns the lock, only the processor holding the lock will update the value of the lock variable.

The need-to-agree in *spinlock* is identified as follows:

1. Instruction level need-to-agree: The CAS is an atomic instruction because all processors need to agree on who gets to perform the swap and thus acquire the lock.
2. Algorithmic need-to-agree: The *spinlock* loops until the CAS is successful.
3. Memory hierarchy need-to-agree: All of the threads are accessing a common lock variable. The cache coherence mechanism enforces agreement on the current value of the lock variable.

Each of these need-to-agree mechanisms can be relaxed one at a time. Doing so yields a suite of variants that can be benchmarked. The difference in the performance of any two variants is due to the difference in the need-to-agree between the two variants.

### 3.3 Eliminating Instruction Level Need-to-agree

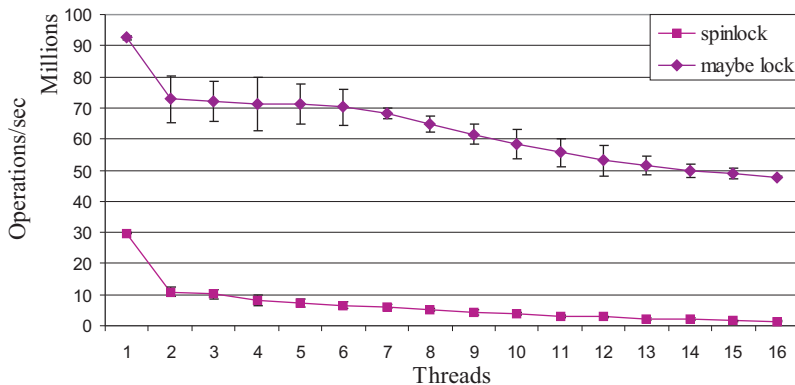
The *spinlock* uses an atomic CAS to prevent the case where two threads examine the lock variable and both see it as `UNLOCKED`. If both threads see the lock variable as `UNLOCKED`, both threads will set the value to `LOCKED` and both will proceed. If two threads are allowed to proceed, mutual exclusion is no longer enforced.

The cost of the instruction level need-to-agree can be measured by replacing the atomic CAS with a non-atomic CAS. The non-atomic CAS variant of *spinlock* is called *maybe lock* because with *maybe lock* a thread might get exclusive access to the lock or it might get shared access. Figure 3 shows the results of the benchmarks for these two variants. The figure shows that *maybe lock* outperforms *spinlock* by as much as a factor of 40 for high thread counts. This benchmark shows that there is a cost to instruction level need-to-agree; however, eliminating this need-to-agree does not allow *maybe lock* to scale.

Both *spinlock* and *maybe lock* show a significant drop going from one to two threads. With only a single thread there is no contention. When there are two or more threads they need to agree with each other. Section 3.5 explains this in more detail.

### 3.4 Eliminating Algorithmic Need-to-agree

The reduced need-to-agree in *maybe lock* can affect performance in two ways: The non-atomic CAS is faster than the atomic CAS, and with the non-atomic CAS there is the potential for increased parallelism because mutual exclusion is not guaranteed. In order to test how much of the increased performance of *maybe*



**Fig. 3.** Comparison of scalability of *spinlock* and *maybe lock*

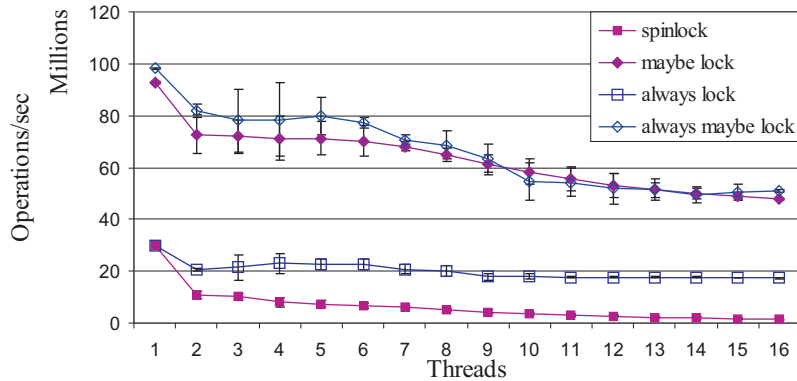
*lock* is due to threads proceeding in parallel, the algorithmic need-to-agree can be relaxed by replacing the `while` with an `if`. This allows all threads to proceed in parallel as if they acquired the lock on the first try. This variant is called *always lock* because the lock always succeeds. The difference in performance between *always lock* with an atomic CAS and with a non-atomic CAS (*always maybe lock*) is due only to the difference in the instruction level need-to-agree.

The results of the benchmarks of these variants are shown in Fig. 4. The performance of *always maybe lock* is significantly better than that of *always lock*. Since neither variant has an algorithmic need-to-agree, the difference in performance is due to instruction level need-to-agree.

The performance of *always lock* is slightly better than that of *spinlock* because *always lock* never spins—it always returns on the first attempt even if the lock is held by another thread. The lower performance of *spinlock* is due to this spinning.

The performance of *always maybe lock* is not significantly different than that of *maybe lock*. *Always maybe lock* does not spin because the algorithmic need-to-agree has been removed. *Maybe lock* may not spin because of the removal of the instruction level need-to-agree. To test this hypothesis, a counter was added to count the number of times the thread spun. The counter for the *always* versions was always zero—these variants never spin. The counter for *maybe lock* indicated that it did spin, but not to the extent of *spinlock*. Some of the performance improvement of *maybe lock* over *spinlock* must be due to reduced spinning caused by the lack of instruction level agreement.

It is interesting to note the *always maybe lock* did not have instruction level need-to-agree nor algorithmic need-to-agree, but it still did not scale. This implies that scalability is limited by memory hierarchy need-to-agree which will be explored in the next section.



**Fig. 4.** Comparison of scalability of *spinlock*, *maybe lock*, *always lock*, and *always maybe lock*

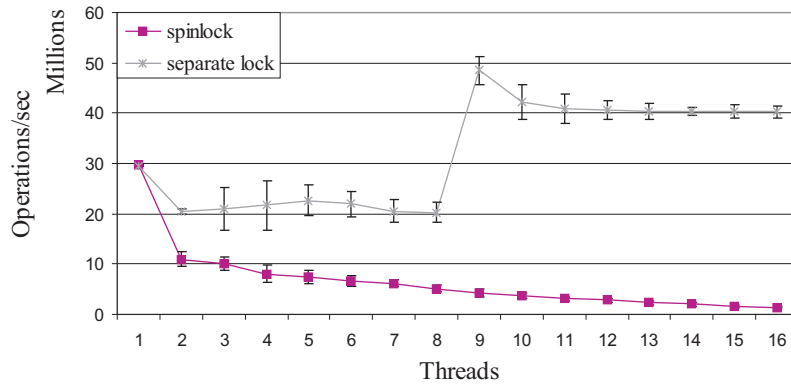
### 3.5 Eliminating Memory Hierarchy Need-to-agree

Memory hierarchy need-to-agree arises because all threads are accessing a common lock variable. The memory hierarchy attempts to get every processor to read the same value from a given memory location even though the most current value might be in a particular processor's cache. *Separate lock* uses a separate lock variable for each thread. *Separate lock* still includes algorithmic need-to-agree and instruction level need-to-agree. However, since each thread uses a separate lock variable, the lock will always be acquired on the first try and there will be no mutual exclusion.

Figure 5 shows a comparison of *spinlock* and *separate lock*. For two to eight threads, there was a slight improvement of *separate lock* over *spinlock*. Even though there was no need-to-agree on the value of the lock variables, *separate lock* still did not scale.

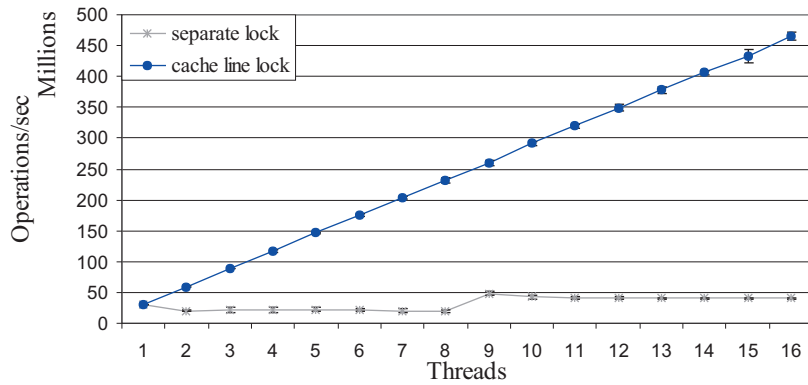
There was a significant improvement in *separate lock* when going from eight to nine threads. Eight lock variables fit in a single cache line. The cache mechanism needs to agree on values with cache line granularity. Threads nine through sixteen used a separate cache line from threads one through eight. Threads one through eight could come to agreement in parallel with threads nine through sixteen. The increased parallelism allowed the performance of *separate lock* for nine to sixteen threads to be roughly double that of one to eight threads.

Because of the cache line implementation, *separate lock* still had a memory hierarchy need to agree even though each thread had a separate lock variable. To eliminate this need to agree, *cache line lock* places each lock variable in a different cache line. Figure 6 shows the performance of *cache line lock* compared with *separate lock*. Not only does *cache line lock* perform better than *separate lock*, but *cache line lock* actually scales.



**Fig. 5.** Comparison of scalability of *spinlock* and *local lock*

It appears that memory hierarchy need-to-agree is the major barrier to scaling. All variants that included memory hierarchy need-to-agree did not scale. *Cache line lock* is the only variant that did not include memory hierarchy need-to-agree and it is the only variant that scaled.



**Fig. 6.** Comparison of scalability of *separate lock* and *cache line lock*

## 4 Benchmark Analysis

The benchmarks showed very little performance dependence on the algorithmic need-to-agree. This may be due to the fact that there was no work performed

between the lock and unlock. As a result, the lock was only held for a short time. If work was done between the lock and unlock, the variants that used the `while` would spin longer while waiting for the lock to be released. The added spinning may have made the algorithmic need-to-agree more evident in the benchmarks.

The instruction-level need-to-agree had a significant impact on algorithm performance but not on scaling. Algorithms that use atomic instructions will still scale provided the atomic instructions access data on separate cache lines. Of course, if the programmer knew that the atomic instructions would always access different memory addresses, there would be no need for atomic instructions. Atomic instructions are only required when there is the possibility that multiple threads will be updating the same memory locations. In other words, instruction level need-to-agree limits performance, but it is memory hierarchy need-to-agree that limits scalability. This is true whether there is true sharing of data or whether there is false sharing due to different data residing in the same cache line.

Algorithmic agreement has been the subject of much work in the literature. Anderson [3] examined *spinlock* with a view to reducing contention for the lock variable. Mellor-Crummey and Scott [4] did similar work and looked at both cache coherent computers and NUMA computers. Fang et. al. [5] proposed a new memory controller concept as an alternative to cache coherence specifically for implementing synchronization primitives.

To generalize the results of the benchmarks, an efficient, scalable synchronization technique should have the following properties:

1. For improved performance, avoid need-to-agree (atomic) instructions where possible.
2. For improved scalability, minimize reliance on a shared variable that gets modified by every operation.

## 5 Beyond *Spinlock*

The following sections look at several common synchronization approaches in the light of the results of the benchmarks presented earlier. In particular, what is the need-to-agree in terms of what atomic instructions are used by the approach and what data gets shared as part of the approach. The performance and scalability implications of each approach will be discussed in terms of its need-to-agree. Section 6 presents benchmark data that validates the analysis and stated implications for performance and scalability.

The approaches can be broadly categorized as follows:

**Pessimistic:** Pessimistic approaches assume that concurrency is bad. As a result, they come to agreement before accessing shared data. The agreement is on who gets to go first.

**Optimistic:** Optimistic approaches assume that in most cases concurrent operations will not conflict with each other. As a result, they come to agreement after accessing shared data. The agreement is on whether there was a conflict or

not. If there was a conflict, the changes to shared data are rolled back and the operation has to be retried.

**Deterministic:** Deterministic approaches allow read-only threads to always succeed without blocking. They are deterministic in that they always succeed, and in that they do so in a fixed number of steps.

### 5.1 Locking—a Pessimistic Approach to Synchronization

Locking is a pessimistic approach to synchronization because it is built on the assumption that concurrency is bad and should be avoided. When a thread requests a lock, it is asking for permission to execute a piece of code (called a critical section). If multiple threads ask for permission at the same time, the semantics of the lock decide which thread(s) are given permission to proceed. When a thread releases a lock, permission may be given to another thread. Each time a thread is granted permission, a need-to-agree operation has to take place.

The most pessimistic approach is mutual exclusion. Only one thread at a time is given permission to enter the critical section. Mutual exclusion suffers from three performance issues:

1. At least one atomic operation per lock/unlock pair.
2. A shared lock variable that gets updated every time the lock is acquired or released.
3. Lack of parallelism.

The first two issues were the specific focus of the benchmarking in Sec. 3. This section will look at the third issue: how to increase parallelism while still using a locking approach.

Fine grained locking partitions the data and has a separate lock for each partition. Fine grained locking allows as many active threads as there are partitions. However, the partitioning comes at a cost. Consider a linked list that is partitioned by having one lock for each node. In order to traverse the list, a lock would have to be acquired and released for each node. The atomic operation and shared lock variable costs would have to be paid for each node. These need-to-agree costs could prevent fine grained locking from scaling even though, in theory, work could be performed in parallel.

Another approach to increasing parallelism is reader-writer locks. Given that readers do not conflict with other readers, multiple readers can be allowed to access the data at the same. Synchronization must guarantee that there is never both a reader and a writer nor multiple writers active at the same time. Mellor-Crummey and Scott [6] present a number of reader-writer lock algorithms. Some of the algorithms they present rely on a single shared lock variable, so they would have the same memory hierarchy need-to-agree problem as a conventional lock. They suggest several approaches that attempt to reduce this need-to-agree by having a thread spin on local data rather than shared data. However, these other approaches still update a shared location for every operation. Spinning happens on a local value, but locking still happens on a shared value. As a result, the memory hierarchy need-to-agree is still present.

The following need-to-agree statements can be made about fine grained locking and reader-writer locking.

1. Fine grained locking has instruction level, algorithmic, and memory hierarchy need-to-agree at the granularity of the lock. For example, in a linked list that has per-node locks, the need-to-agree costs must be paid for each node that is accessed.
2. Reader-writer locking has instruction level, algorithmic, and memory hierarchy need-to-agree. In order for readers to proceed in parallel, each reader must pay the need-to-agree cost before proceeding.

Because both of these algorithms include memory hierarchy need-to-agree they are not expected to scale even though they claim to allow parallelism.

## 5.2 Nonblocking Synchronization—an Optimistic Approach to Synchronization

One of the problems with a locking approach to synchronization is that if a thread fails while holding a lock, the lock may never get released. This has the potential of causing deadlock where all other threads eventually block. There has been much research into synchronization techniques that avoid this potential deadlock problem [7–11]. Many of the proposed solutions can be considered optimistic approaches to synchronization. Unlike locking approaches where a thread needs permission to enter a critical section, all threads are allowed to enter the critical section. In order to prevent data corruption in the event of concurrent updates, optimistic approaches require a thread to get permission not to start an update, but to complete (or commit) an update.

The general approach to non-blocking synchronization can be summarized by the following steps

1. Save the state of the data structure.
2. Make a change in thread local memory.
3. Commit or rollback: If the state is unchanged, make the change visible in public memory. If the state has changed, rollback the change and try again

Note that steps 1 and 3 are need-to-agree steps and typically involve atomic instructions. If an algorithm strictly follows these steps, it is easy to see that progress guarantees can be made. The only way a given thread will not make progress is if it rolls back and retries in step 3. But in order for the rollback to happen, some other thread must have changed the state of the data—in other words another thread must have made progress by committing in step 3.

Not all non-blocking synchronization algorithms strictly follow the steps given above. In some cases, the data structure gets modified as part of step 1. For these algorithms, the modifications must leave enough information so that another thread can either complete or rollback the operation. Allowing other threads to complete operations allows progress guarantees to be made in the presence of failed or stalled threads.

Non-blocking synchronization algorithms tend to be complicated. They need to make both data safety guarantees and progress guarantees for all possible interleavings of threads. Each algorithm tends to be specific for a particular data structure [9]. Rather than focussing on non-blocking synchronization in general, this paper focuses on a particular algorithm. Since nonblocking synchronization techniques often follow similar patterns, the analysis of this particular algorithm can be applied to other nonblocking synchronization techniques.

Fomitchev and Rupert [9] describe a lock-free algorithm for sorted linked lists. Their algorithm provides search, insert, and delete functions. The main complication in their algorithm is handling deletes. There are two issues that must be correctly handled. First, the algorithm must guarantee that the node being deleted does not have its link pointer updated concurrently with the deletion. Second, the algorithm must guarantee that the memory for the deleted node is not reclaimed while concurrent readers are still accessing the node.

To illustrate the first case, consider a linked list with three consecutive nodes labeled *Previous*, *Delete*, and *Next*. The node *Delete* is the node that is going to be deleted. The algorithm must guarantee that a node *New* does not get inserted between *Delete* and *Next* at the same time that node *Delete* gets deleted. The insert causes *Delete* to point to *New*. The delete causes *Previous* to point to *Next* bypassing *Delete* and more importantly, bypassing the inserted node *New*.

For the second case, the node *Delete* might get removed from the list while another thread is still accessing the node. If the thread that performs the deletion frees the memory and the memory gets reused while the other thread is still accessing it, data corruption or program failure may result.

Fomitchey and Rupert deal explicitly with the first issue but just cite other references [12] for the second issue. Using the node names from above, the delete process happens as follows:

1. *Previous* gets flagged indicating that the node after *Previous* is going to be deleted.
2. *Delete* gets marked for deletion.
3. *Delete* gets physically removed from the list by having *Previous* point to *Next*.

Each of these are need-to-agree steps and are performed using a compare and swap.

Any thread that accesses a marked or flagged node will first complete the deletion. After completing the deletion, the thread will have to rescan the list. Rather than beginning the rescan at the beginning of the list, Fomitchey and Rupert's algorithm uses back-link pointers to find the first predecessor that is not a candidate for deletion.

Inserts are simpler than deletes because there is no danger of the new node being updated concurrently with the insert (the new node is not visible to others until after the insert). The insert is performed with a single compare and swap of the link pointer in the preceding node. The compare and swap prevents concurrent inserts at the same location. One compare and swap will succeed, the other will fail. The compare and swap also prevents inserts following a node

being deleted. The mark or flag mentioned above will cause the compare and swap to fail.

The following need-to-agree statements can be made concerning Fomitchey and Rupert's algorithm:

1. Reads in the absence of concurrent updates do not require any atomic instructions and do not update any shared data.
2. Inserts in the absence of any concurrent operations require a single atomic instruction and update a single shared location.
3. Deletes in the absence of any concurrent operations require three atomic instructions and update two shared locations. One of the two locations gets updated twice.

Based on these statements, the following predictions can be made on scalability:

1. Reads in the absence of updates should scale.
2. Updates will interfere with the scalability of reads.
3. Concurrent updates of different locations may scale if the updated locations do not share a cache line.
4. Concurrent updates of the same location (or multiple locations in the same cache line) will not scale.

### 5.3 Relativistic Programming—a Deterministic Approach to Synchronization

Relativistic programming is a synchronization technique that focuses specifically on increased scalability. Relativistic programming can be characterized by the following two properties [13]:

1. Tolerance of the lack of a global order on events.
2. Tolerance of conflicts between concurrent threads.

In other words, relativistic programming requires less agreement on the order of events and on who can access the data.

Most other synchronization techniques attempt to enforce a global order on all events. This global ordering is often used as part of a proof of correctness for the technique [14, 15]. Adherents to relativistic programming claim that global ordering is a more strict requirement than is often required for correctness [16]. The global ordering also comes at a performance cost [15].

Most synchronization techniques attempt to either prevent conflicts (mutual exclusion) or resolve conflicts by causing some threads to repeat work (non-blocking synchronization). Relativistic programming tolerates conflicts in the form of concurrent reads and writes to the same data.

Both of the properties of relativistic programming are typically implemented by versioning memory and allowing multiple versions to exist at the same time. To illustrate, consider Read/Copy-Update (RCU) [17]. The principle behind RCU is that threads that update data make a copy of the data they are going to change, update the copy, then atomically make the change visible. When

applied to linked lists, if a thread wants to update the contents of a node in the list, the thread makes a copy of the node. Updates are applied to the copy while readers continue to see the old node. Once the updates are complete, the writer swaps pointers so the new node is visible in the list instead of the old node. At this point in time, new readers will access the new node. Existing readers may still be accessing the old node. Once all existing readers have completed, the old node can be freed.

RCU does not specify how writers synchronize with each other. For example, writers could use mutual exclusion between writers or non-blocking synchronization could be used between writers. In either case, readers do not participate in the synchronization: they proceed unimpeded.

The need-to-agree for RCU can be summarized as follows:

1. Readers require no atomic instructions and do not modify any shared data.
2. Readers have no need-to-agree with writers.
3. The scaling of writers is limited by the technique used to synchronize between writers.

If mutual exclusion is used between writers, the following predictions can be made for RCU scaling:

1. Readers should scale linearly even in the presence of writers.
2. Writers will not scale.
3. The presence of concurrent readers should have minimum impact on the performance of writers.

## 6 Performance of Linked Lists with Various Synchronization Techniques

This section presents preliminary benchmark results for linked list implementations using the following four synchronization methods: spinlock, reader-writer lock (RWL), non-blocking synchronization (NBS) and read/copy-update (RCU). The list supported lookups, inserts, and deletes. Lookups are considered read operations. Inserts and deletes are considered write operations.

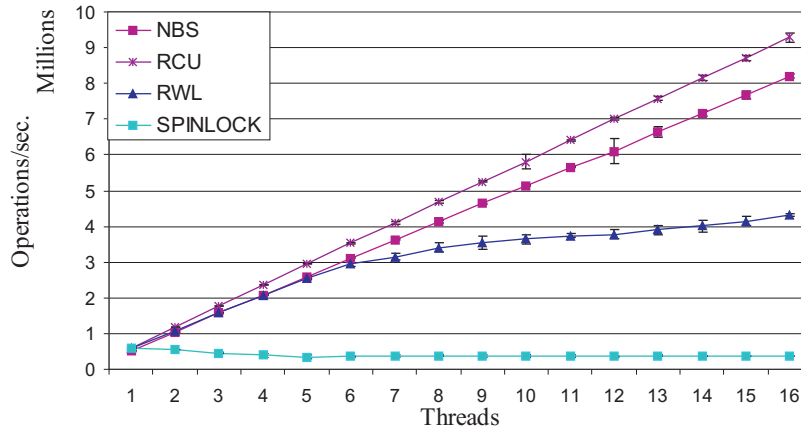
The algorithms used in the benchmarks were derived from the following sources: The spinlock uses the algorithm presented in Sec. 3.2. The RWL code was derived from pseudocode presented by Mellor-Crummey and Scott [6]. The NBS code was developed from pseudocode presented by Fomitchev and Ruppert [9]. The user mode RCU code was derived from unpublished code by Paul McKenney. McKenney referred to his own code as a “toy” implementation. Substantial modifications were made to the code to remove some of the “toy” considerations. All of the algorithms used the Google’s tcmalloc memory allocator [18] because it fulfills most of the requirements for safe reclamation for the NBS implementation.

### 6.1 Read Performance

The first benchmark examines read performance of the various algorithms. For this benchmark, all threads were readers. No updates were performed. Based on the need-to-agree, it is expected that RCU should have high performance and scale well because there is no need-to-agree in the read path. With NBS, there is no need-to-agree unless there are updates so in the read only case NBS should scale. Since the NBS algorithm is more complicated than RCU, performance is expected to be lower. RWL should allow readers to proceed in parallel. However, there is a need-to-agree on the memory location of the lock variable. As a result, RWL is not expected to scale linearly. Spinlock is not expected to scale because it is a mutual exclusion algorithm that only allows one thread at a time to proceed.

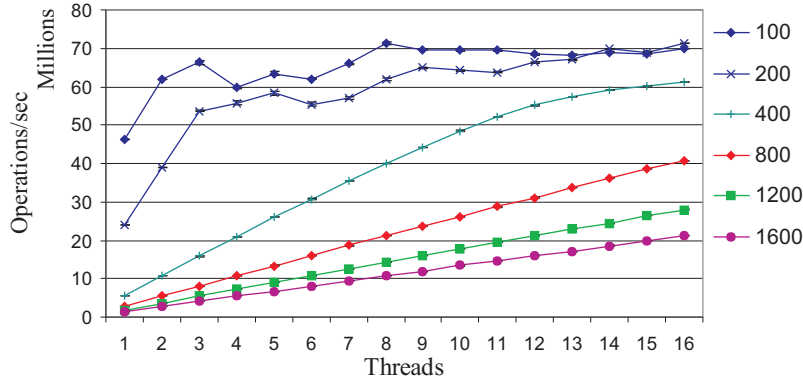
The results of this benchmark are shown in Fig. 7. The results confirm the expectations. RCU and NBS both scaled. RCU outperformed NBS. RWL scaled, but not to 16 processors. Spinlock scaled negatively.

The RWL case did not scale all the way to 16 processors because the need-to-agree on the lock variable acted as a sequential bottleneck. To test this hypothesis, a number of benchmarks were run with with varying list lengths. Since on average half the list has to be searched to find an item, the average amount



**Fig. 7.** Comparison of read performance in the absence of writers for spinlock, RWL, RCU, and NBS.

of work done per lock is proportional to the list length. Tests were run with lists varying in length from 100 through 1600 elements. The results are given in Fig. 8. The key shows the length of the list for each run. As the length of the list increased, the algorithm scaled over a larger number of processors. Lists of up to 200 elements scaled to only three processors. Lists of 800 or more elements appeared to scale to at least 16 processors.



**Fig. 8.** Scalability of Reader Writer Locking for varying list lengths. The key shows the list length.

## 6.2 Write Performance

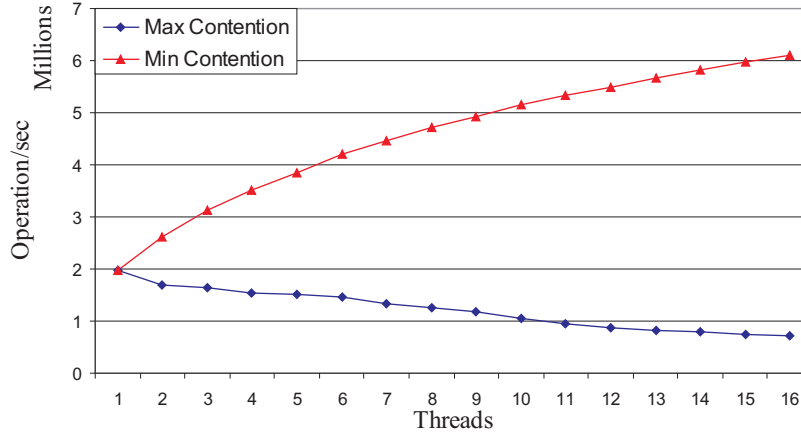
Of all the algorithms, only NBS is expected to scale on the write side. The other algorithms use mutual exclusion between writers<sup>4</sup> so only one writer at a time is allowed. Two tests were run to determine the scalability of NBS writers. The first test had all the threads repeatedly insert and delete the same element. This is the maximum contention case and should show the worst case performance. It is not expected to scale because all the threads need to agree on the value of the pointer they are updating when inserting and deleting the node. The second benchmark in this set has each thread inserting and deleting a node as follows:

$\text{thread}_i$  inserts/deletes node with key  $X_i$   
 $X_i \neq X_j$  if  $i \neq j$   
 $X_i$  and  $X_j$  are never in adjacent nodes

These conditions should provide minimum contention and should provide maximum scaling. They are analogous to *cache line lock* from Sec. 3.5 where each thread is updating a value in a separate cache line.

The results for these benchmarks are presented in Fig. 9. As expected the maximum contention case scales negatively because of the need-to-agree on the nodes being inserted and deleted. The minimum contention scales, but not linearly. The performance gains appear to be tapering off as more processors are added. There are several reasons for this. Threads that are updating nodes later

<sup>4</sup> The lack of parallelism on the write side is an artifact of the particular implementations that were benchmarked not of the algorithms themselves. There are implementations of the other algorithms that allow parallelism on the write side. Fine grained locking should allow parallelism on the write side and RCU can be implemented with fine grained locking on the write side or even NBS on the write side.



**Fig. 9.** Comparison of write performance for maximum and minimum contended cases of non-blocking synchronization.

in the list will encounter cache misses due to the updated nodes earlier in the list (memory hierarchy need-to-agree), and threads that are updating nodes later in the list will periodically encounter a partial update early in the list and complete the update before continuing (algorithmic need-to-agree). This later case has two negative side effects: it slows the current thread because it completes work for another thread. It also slows the thread that initiated the update because the thread that completes the update gets write access to the cache line containing the data. The thread that initiated the update will incur a cache miss when it attempts to finish the update.

## 7 Conclusion

We have shown that an algorithm’s need-to-agree can be used to predict its performance and scalability. The different forms of need-to-agree affect performance and scalability in different ways.

**Instruction level need-to-agree:** Instructions that require agreement between processors are slower than instructions that do not require such agreement. This implies that algorithms which depend on these instructions will be slower than algorithms which do not. However, the use of need-to-agree instructions does not necessarily limit scalability as was seen by the *cache line lock* and *reader-writer lock* readside scalability benchmarks.

**Algorithmic need-to-agree:** Algorithmic need-to-agree can limit scalability by limiting parallelism (*spinlock*), or by increasing the amount of work that a thread has to perform (*non-blocking synchronization*).

**Memory hierarchy need-to-agree:** Memory hierarchy need-to-agree appears to be the major limiting factor in scalability. The non-uniform nature of memory means that a cache coherence mechanism is required to provide agreement on a value stored in memory. The impact of this need-to-agree is not limited to programs that share actual memory locations. Sharing locations within the same cache line is sufficient to cause scalability problems.

The work presented in this paper has two main implications:

1. Algorithms that purport to allow parallelism do not necessarily lead to scalable solutions. For example, *reader-write locks* should allow parallelism for read only workloads. However, if the amount of work performed inside the critical section is small, the algorithm will not scale because the need-to-agree is a sequential bottleneck. This observation is particularly relevant for multiprocessor operating systems. The amount of work within a system call tends to be small. A synchronization technique with a high need-to-agree is not likely to scale even if the technique claims to allow parallelism.
2. Modern multicore computers behave like NUMA computers where there is a non-trivial cost to accessing non-local data. But unlike traditional NUMA computers, modern multicore computers do not give programmers explicit control over data placement nor data migration. This suggests that the memory abstraction that is currently offered on these multicore computers may not be the optimal abstraction for writing efficient scalable programs. Future work should examine this abstraction to determine if there would be performance and scalability benefits to providing the programmer with more explicit control of when and how cache coherence communication takes place.

**Acknowledgments** I want to thank Jonathan Walpole for providing both the direction necessary to succeed in this research and also the freedom which allows one to fail. He also provided a continual stream of questions that needed answering. I also want to thank Paul E. McKenney for giving insight into the user mode RCU implementation that was used in the benchmarks and for doing some of the early work in relativistic programming that helped frame the question, “What do we really need to agree on, anyway?”. Finally, I want to thank Josh Triplett for valuable input on editing this paper.

Funding for this research was provided by the National Science Foundation under Grant No. CNS-0719851.

## References

1. Sutter, H., Larus, J.: Software and the concurrency revolution. *Queue* **3**(7) (2005) 54–62
2. Intel. In: Intel 64 and IA-32 Architectures Software Developer’s Manual. Volume 2A. Intel (March 2009) 3.168–3.173
3. Anderson, T.E.: The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst.* **1**(1) (1990) 6–16

4. Mellor-Crummey, J.M., Scott, M.L.: Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.* **9**(1) (1991) 21–65
5. Fang, Z., Zhang, L., Carter, J.B., Cheng, L., Parker, M.: Fast synchronization on shared-memory multiprocessors: an architectural approach. *J. Parallel Distrib. Comput.* **65**(10) (2005) 1158–1170
6. Mellor-Crummey, J.M., Scott, M.L.: Scalable reader-writer synchronization for shared-memory multiprocessors. In: PPOPP '91: Proceedings of the third ACM SIGPLAN symposium on Principles and practice of parallel programming, New York, NY, USA, ACM (1991) 106–113
7. Fraser, K., Harris, T.: Concurrent programming without locks. *ACM Trans. Comput. Syst.* **25**(2) (2007) 5
8. Herlihy, M., Luchangco, V., Moir, M.: Obstruction-free synchronization: Double-ended queues as an example. In: ICDCS '03: Proceedings of the 23rd International Conference on Distributed Computing Systems, Washington, DC, USA, IEEE Computer Society (2003) 522
9. Fomitchev, M., Ruppert, E.: Lock-free linked lists and skip lists. In: PODC '04: Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing, New York, NY, USA, ACM (2004) 50–59
10. Michael, M.M.: High performance dynamic lock-free hash tables and list-based sets. In: SPAA '02: Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures, New York, NY, USA, ACM (2002) 73–82
11. Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: PODC '96: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing, New York, NY, USA, ACM (1996) 267–275
12. Michael, M.M.: Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In: PODC '02: Proceedings of the twenty-first annual symposium on Principles of distributed computing, New York, NY, USA, ACM (2002) 21–30
13. McKenney, E., P., Walpole, J.: Definition of relativistic programming. email exchange (April 2009)
14. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.* **12**(3) (1990) 463–492
15. Attiya, H., Welch, J.L.: Sequential consistency versus linearizability. *ACM Trans. Comput. Syst.* **12**(2) (1994) 91–122
16. Schiper, A., Birman, K., Stephenson, P.: Lightweight causal and atomic group multicast. *ACM Trans. Comput. Syst.* **9**(3) (1991) 272–314
17. McKenney, P.E., Walpole, J.: What is rcu, fundamentally? <http://lwn.net/Articles/262464/> (December 2007)
18. Ghemawat, S., Menage, P.: Tcmalloc : Thread-caching malloc. <http://goog-perftools.sourceforge.net/doc/tcmalloc.html>