White Paper

Is the Free Lunch Really Over? Scalability in Manycore Systems Part 2: Using Locks Efficiently

By Michael Wrinn

Introduction

For the multi-core, shared-memory computing platforms currently, applications are typically rendered parallel with one or another choice of threading model – extensions, such as Pthreads or OpenMP™, to sequential languages. While this situation is less than ideal (see the analysis, for example, by Lee, 2006), it is the current industry practice. The shared-state paradigm of typical threading models, where all threads share the same memory with equal access to its data, requires careful protocol to prevent unintended consequences. The key difficulty is the *race* condition, where threads access data in an unanticipated sequence, with unintended results. Race conditions are the worst kind of bug: insidious, very difficult to detect, reproduce, or diagnose. Tools are essential to mitigating these difficulties [shameless plug: the Intel® Thread Checker is quite good at this]. Another category of irritants is deadlocks/livelocks, though – due to the fact that the code execution halts – these are normally easier to diagnose and correct. These correctness challenges are widely discussed in threading books and articles, and will not be addressed here. We focus, instead, on a necessary component of threaded programming – locks – and their potential impact on performance.

Contents

Figures

Introduction

For the multi-core, shared-memory computing platforms currently, applications are typically rendered parallel with one or another choice of threading model – extensions, such as Pthreads or OpenMP™, to sequential languages. While this situation is less than ideal (see the analysis, for example, by Lee, 2006), it is the current industry practice. The shared-state paradigm of typical threading models, where all threads share the same memory with equal access to its data, requires careful protocol to prevent unintended consequences. The key difficulty is the race condition, where threads access data in an unanticipated sequence, with unintended results. Race conditions are the worst kind of bug: insidious, very difficult to detect, reproduce, or diagnose. Tools are essential to mitigating these difficulties [shameless plug: the Intel® Thread Checker is quite good at this]. Another category of irritants is deadlocks/livelocks, though – due to the fact that the code execution halts – these are normally easier to diagnose and correct. These correctness challenges are widely discussed in threading books and articles, and will not be addressed here. We focus, instead, on a necessary component of threaded programming – locks – and their potential impact on performance.

Even when parallel programs have been scrubbed to the point of reasonable confidence in their correctness, there remain potential unintended consequences in performance degradation. Applying a lock does, after all "re-serialize" that portion of execution; the goal then is to use these only as much locking as is necessary, but no more.

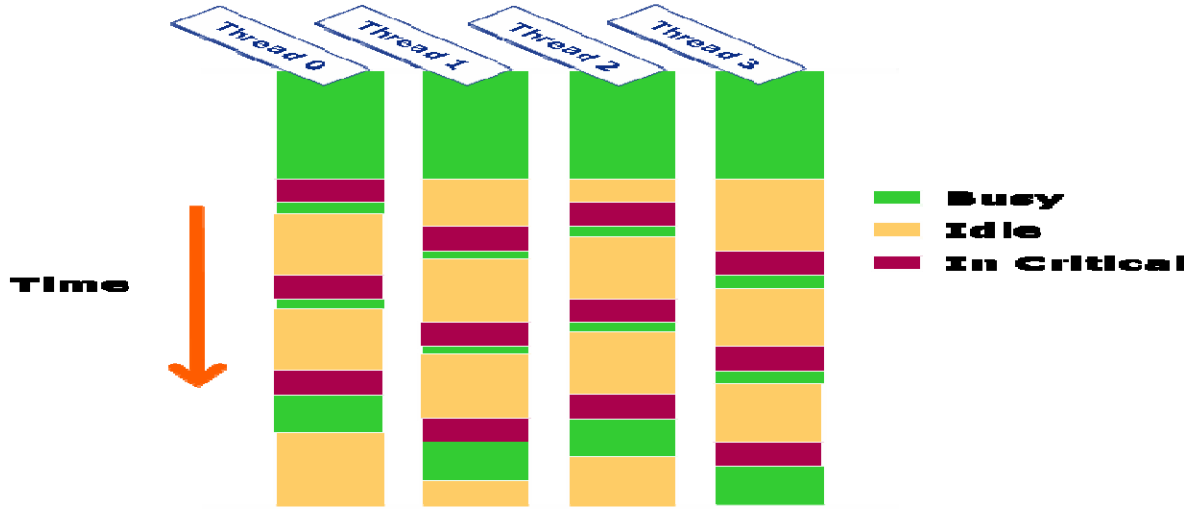
This scalability series assumes the code is correct, and looks are factors which may inhibit the scalability of the correctly-running application. In this episode, we look at the efficient use of locks, and in particular at two guiding principles: lock the data, not the code, and use the right lock for the job.

Factors impacting scalability: using locks efficiently

1.1 Profile view: when locks intrude

Good lock management is essential to scalable parallel performance. Figure 2-1 illustrates the opposite, a profile trace of poor lock management. Each bar represents a timeline of execution for a particular thread: green areas indicating *busy*, executing presumably useful work; *idle*, as the term suggests, is time spent waiting for other threads, and *in critical* indicates time spent inside locked regions of execution. (This example is from an actual profile, illustrated using the Intel® Thread Profiler.) In the example profile shown, once a critical region is reached by one thread, all others are rendered idle; the overall execution in those sections has been serialized. This particular problem often occurs when work inside and outside the protected region (e.g. a critical section) is very small, so threads "pile up" on the lock. Such design limitations may be avoided by following the guidelines discussed below.

Figure 2-1. Thread performance degradation from inefficient locking mechanisms.



1.2 Lock data, not code

Lock only what needs to be accessed serially. What usually need this kind of protection are data elements; whenever possible, **lock data, not code**. Programming adepts have been repeating this for years, and it bears repeating again and again. (see, for example, Russell's somewhat salty and entertaining Unreliable Guide to Locking, published in 2000. More recently, Sutter nicely alludes to this technique (without actually repeating the phrase) in a 2008 article on concurrency-friendly data structures.)

In our own courses at Intel Software College, we teach this concept right up front, using a link-list example. Consider the example of creating a hash table (a collection of linked lists), shown schematically in Figure 2-2. The lists may be operated upon in parallel, but

updates to any particular list must be protected (locked) to prevent a race condition. One very simple way to accomplish this is with a Critical Section, shown (with OpenMP syntax) in Figure 2-3.

Figure 2-2. Schematic representation of hash table creation.

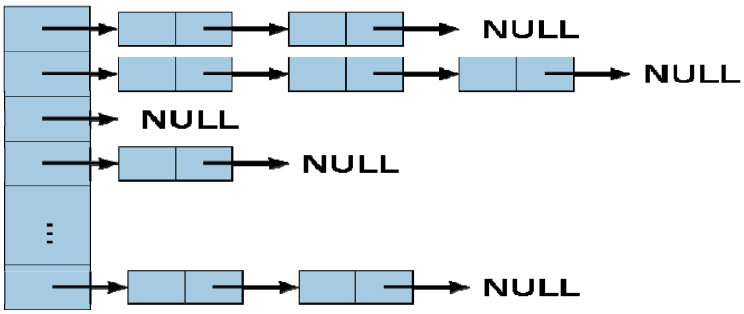


Figure 2-3. Code segment showing inefficient use of Critical Section (don't do this).

```
#pragma omp parallel for private (index)
for (i = 0; i < elements; i++) {
   index = hash(element[i]);
   #pragma omp critical
   insert_element (element[i], index);}
```

In this example, we have locked the *code*: we prevent two threads from trying to update the same list simultaneously by putting function insert_element inside a critical section. Unfortunately, this effectively serializes the loop, because the work of inserting an element into the hash table is the principal activity of the loop. This approach will result in little, if any, speed improvement from parallelism.

A much better approach is to lock the *data*. OpenMP, as with most threading models, gives us the ability to lock individual chains of the hash table. We associate a lock variable with each index in the hash table. When we are ready to insert an element into a particular chain, we set the lock associated with that chain. The advantage of this approach is that a group of threads may all be inserting elements into the hash table in parallel, as long as these elements hash to different table indices.

To use this approach, a bit of housekeeping is in order: the lock variable must be declared as type omp_lock_t, and initialized with the function omp_lock_t. Figure 2-4a shows these steps for the hash table case, where we have declared a lock variable hash_lock.

Figure 2-4a. Code segments showing housekeeping steps to use omp_set_lock.

```
/* hash_lock declared as type omp_lock_t */
omp_lock_t hash_lock[HASH_TABLE_SIZE];

/* locks initialed in function 'main' */
for (i = 0; i < HASH_TABLE_SIZE; i++)

              omp_init_lock(&hash_lock[i]);
```

Figure 2-4b. Code segment showing efficient locking with omp_set_lock (do this).

```
void insert_element (ELEMENT e, int i)
{
   omp_set_lock (&hash_lock[i]);
   /* Code to insert element e */
   omp_unset_lock (&hash_lock[i]);
}
```

1.3 Use the right locking mechanism for the job

Using the right locking mechanism nearly always implies: use the *lightest possible* mechanism. Threading models supply a hierarchy of locks; in the Win32 API, for example, the choices of synchronization primitives are as follows, in increasing order of execution cost:

- Atomic increments/decrements
    - InterlockedIncrement
- Critical Section, Critical Section with spin count
    - EnterCriticalSection, LeaveCriticalSection, SetCriticalSectionSpinCount
    - Works within a single process
- Events
    - Signal that a condition has been changed or satisfied
- Mutex
    - Works both within and across processes (since it's a kernel object)
- Semaphore
    - also works across processes

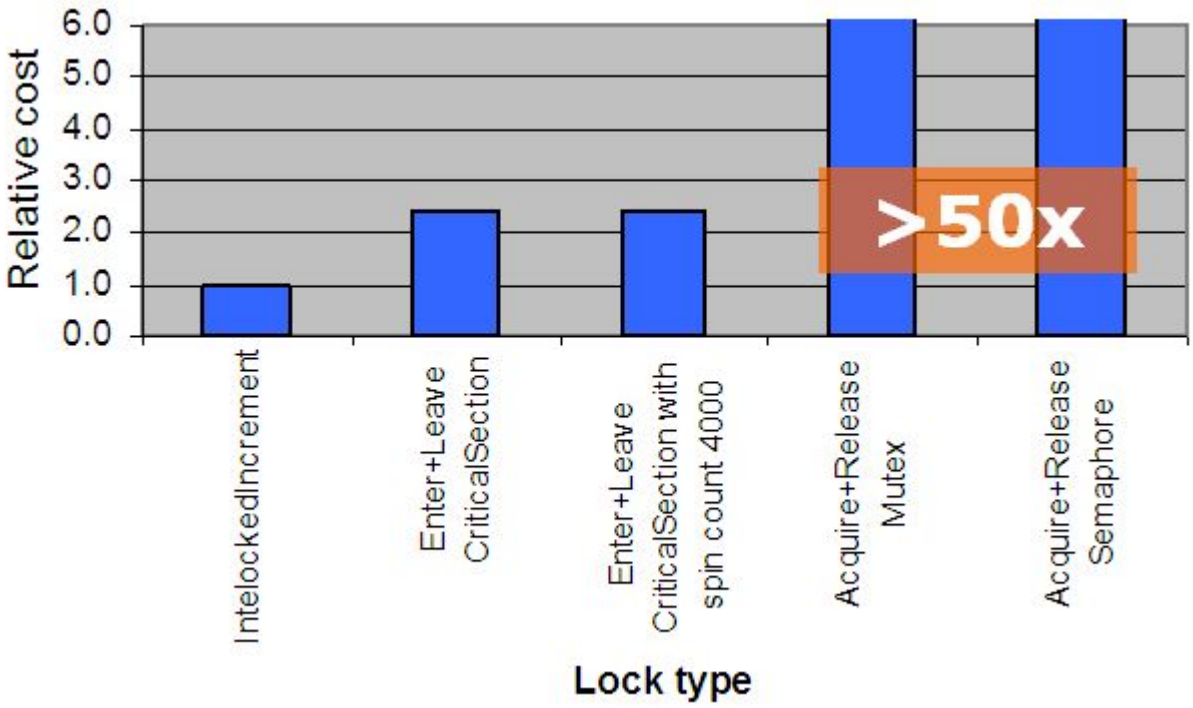
The "spin count" variant of critical section can work well in highly-contended situations: the critical section is initialized with a count; whenever a thread attempts to access that section,

rather than sleeping, it remains active for a time period determined by the spin count, and then tests again. This avoids the overhead of blocking, then reawakening, the thread which is waiting for a lock to be released.

Measurements of the overhead (the actual lock times, normalized to the smallest time of InterlockedIncrement) of each locking mechanism are compared in Figure 2-5. The experiment was arranged to avoid contention (thus the critical section shows the same performance with and without spin count). Notice that the mutex and semaphore locks are dramatically more expensive, more than 50 times slower than the others, going right off the scale of the graph. To repeat: in terms of cost:

Mutex or semaphore >> critical section > InterlockedIncrement

Figure 2-5. Relative cost for different locking mechanisms, scaled to InterlockedIncrement.

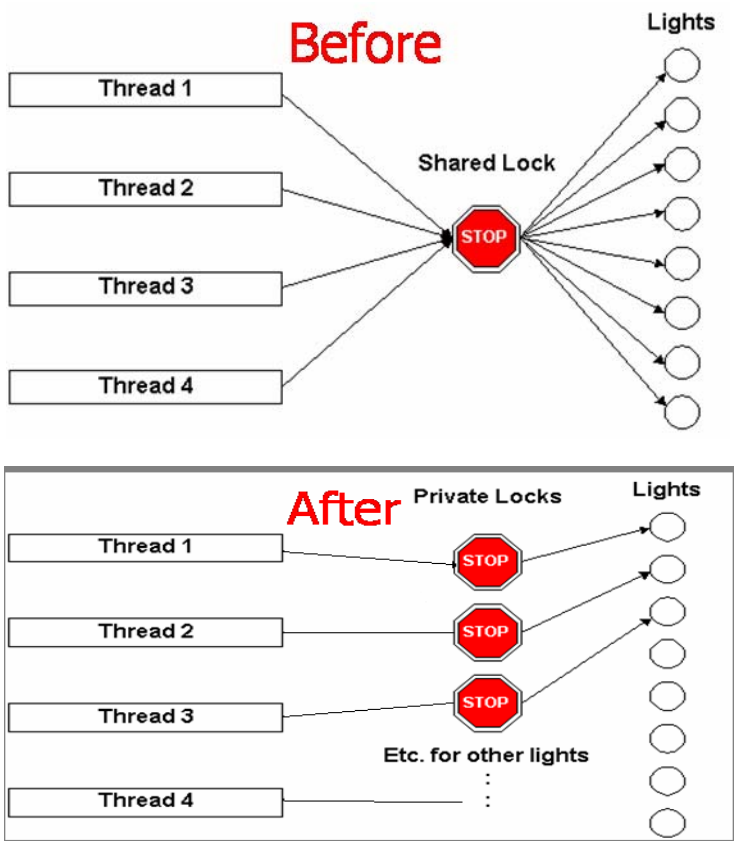


For those who'd like to try this at home: the complete lab which generated the results of Figure 2-5, including course code, is available for college/university classroom through Intel's Academic Community.

Besides choosing the right level of lock – that is, the least expensive lock which will do the job – it is important to apply the lock in the least intrusive way possible. In generally, global (static) locks are to be avoided. A simple case study illustrates this point. A client code was found to show *negative* scaling at 4 threads or more – that is, the application ran slower with four threads, on four cores, than it did when running with a single thread. The root cause was discovered to be that a class defined a critical section as a *static* member variable. The solution: have each instance of class use separate lock by removing *static* declaration. Once this was done, the application performance did

increase with increasing core/thread count. Figure 2-6 shows the concept in schematic form, before and after this change.

Figure 2-6. Schematic comparison of static (Before) to private locks (After).



To **conclude**: efficient lock usage has a dramatic impact on scalability. Lock data, not code. Use the least expensive locking mechanism possible. Beware of static locks.

References

Edward Lee, The Problem with Threads, IEEE Computer, May 2006

Intel Threading Analysis Tools

Paul Rusty Russell, Unreliable Guide To Locking

Herb Sutter, Dr Dobbs Journal 33(7), July 2008

Intel Software College. Contains links to classes such as *Introduction to Parallel Programming* and more.

A growing body of teaching material for concurrency may be found at the Intel Academic Community.

A.1.1 Acknowledgments

This material is derived from a longer session by Intel Software College.

About the Author

Michael Wrinn is a senior course architect in the Intel Software College, where he collaborates with universities to bring parallel computing to the mainstream of undergraduate education. Prior assignments include managing Intel's software engineering lab in Shanghai, and directing the human interface technology research lab. He was Intel's representative to the committee which produced the first OpenMP specification, and remains active in the parallel computing community. Before joining Intel, Michael worked at Accelrys (San Diego), implementing commercial and research simulation codes on a wide variety of parallel/HPC systems. He holds a Ph.D. (in quantum mechanics) and a B.Sc. (mathematics/chemistry/physics) from McGill University.

§