Use Thread-local Storage to Reduce Synchronization

<keywords: threading, thread-local storage, synchronization, OpenMP, Pthreads, Win32 threads>

Abstract

Synchronization is often an expensive operation that can limit the performance of a multi-threaded program. Using thread-local data structures instead of data structures shared by the threads can reduce synchronization in certain cases, allowing a program to run faster.

This article is part of the larger series, "The Intel Guide for Developing Multithreaded Applications," which provides guidelines for developing efficient multithreaded applications for Intel® platforms.

Background

When data structures are shared by a group of threads and at least one thread is writing into them, synchronization between the threads is sometimes necessary to make sure that all threads see a consistent view of the shared data. A typical synchronized access regime for threads in this situation is for a thread to acquire a lock, read or write the shared data structures, then release the lock.

All forms of locking have overhead to maintain the lock data structures, and they use atomic instructions that slow down modern processors. Synchronization also slows down the program, because it eliminates parallel execution inside the synchronized code, forming a serial execution bottleneck. Therefore, when synchronization occurs within a time-critical section of code, code performance can suffer.

The synchronization can be eliminated from the multithreaded, time-critical code sections if the program can be re-written to use thread-local storage instead of shared data structures. This is possible if the nature of the code is such that real-time ordering of the accesses to the shared data is unimportant. Synchronization can also be eliminated when the ordering of accesses is important, if the ordering can be safely postponed to execute during infrequent, non-time-critical sections of code.

Consider, for example, the use of a variable to count events that happen on several threads. One way to write such a program in OpenMP* is as follows:

```c
int count=0;
#pragma omp parallel shared(count)
{
    . . .
    if (event_happened) {
        #pragma omp atomic
        count++;
    }
    . . .
}
```

This program pays a price each time the event happens, because it must synchronize to guarantee that only one thread increments `count` at a time. Every event causes synchronization. Removing the synchronization makes the program run faster. One way to do this is to have each thread count its own events during the parallel region and then sum the individual counts later. This technique is demonstrated in the following program:
This program uses a `tcount` variable that is private to each thread to store the count for each thread. After the first parallel region counts all the local events, a subsequent region adds this count into the overall count. This solution trades synchronization per event for synchronization per thread. Performance will improve if the number of events is much larger than the number of threads. Please note that this program assumes that both parallel regions execute with the same number of threads. The call to `omp_set_dynamic(0)` prevents the number of threads from being different than the number requested by the program.

An additional advantage of using thread-local storage during time-critical portions of the program is that the data may stay live in a processor’s cache longer than shared data, if the processors do not share a data cache. When the same address exists in the data cache of several processors and is written by one of them, it must be invalidated in the caches of all other processors, causing it to be re-fetched from memory when the other processors access it. But thread-local data will never be written by any other processors than the one it is local to and will therefore be more likely to remain in the cache of its processor.

The code snippet above shows one way to specify thread-local data in OpenMP. To do the same thing with Pthreads, the programmer must create a key for thread-local data, then access the data via that key. For example:

```c
#include <pthread.h>

pthread_key_t tsd_key;
<arbitrary data type> value;

if( pthread_key_create(&tsd_key, NULL) ) err_abort(status, “Error creating key”);
if( pthread_setspecific( tsd_key, value))
  err_abort(status, “Error in pthread_setspecific”);
  ...
value = (<arbitrary data type>)pthread_getspecific( tsd_key );
```
With Windows threads, the operation is very similar. The programmer allocates a TLS index with TlsAlloc, then uses that index to set a thread-local value. For example:

```c
DWORD tls_index;
LPVOID value;

tls_index = TlsAlloc();
if (tls_index == TLS_OUT_OF_INDEXES) err_abort(tls_index, "Error in TlsAlloc");
status = TlsSetValue(tls_index, value);
if (status == 0) err_abort(status, "Error in TlsSetValue");
value = TlsGetValue(tls_index);
```

In OpenMP, one can also create thread-local variables by specifying them in a `private` clause on the `parallel` pragma. These variables are automatically deallocated at the end of the parallel region. Of course, another way to specify thread-local data, regardless of the threading model, is to use variables allocated on the stack in a given scope. Such variables are deallocated at the end of the scope.

**Advice**

The technique of thread-local storage is applicable if synchronization is coded within a time-critical section of code, and if the operations being synchronized need not be ordered in real-time. If the real-time order of the operations is important, then the technique can still be applied if enough information can be captured during the time-critical section to reproduce the ordering later, during a non-time-critical section of code.

Consider, for example, the following example, where threads write data into a shared buffer:

```c
int buffer[NENTRIES];

main() {
    . . .

    #pragma omp parallel
    {
        . . .
        update_log(time, value1, value2);
        . . .
    }
    . . .
}

void update_log(time, value1, value2)
{
    #pragma omp critical
    {
        if (current_ptr + 3 > NENTRIES)
        { print_buffer_overflow_message(); }

        buffer[current_ptr] = time;
        buffer[current_ptr+1] = value1;
        buffer[current_ptr+2] = value2;
    }
```
current_ptr += 3;
}

Assume that `time` is some monotonically increasing value and the only real requirement of the program for this buffer data is that it be written to a file occasionally, sorted according to time. Synchronization can be eliminated in the `update_log` routine by using thread-local buffers. Each thread would allocate a separate copy of `tpbuffer` and `tpcurrent_ptr`. This allows the elimination of the critical section in `update_log`. The entries from the various thread-private buffers can be merged later, according to the time values, in a non-time-critical portion of the program.

**Usage Guidelines**

One must be careful about the trade-offs involved in this technique. The technique does not remove the need for synchronization, but only moves the synchronization from a time-critical section of the code to a non-time-critical section of the code.

- First, determine whether the original section of code containing the synchronization is actually being slowed down significantly by the synchronization. Intel® Parallel Amplifier and/or Intel® VTune™ Performance Analyzer can be used to check each section of code for performance problems.

- Second, determine whether the time ordering of the operations is critical to the application. If not, synchronization can be removed, as in the event-counting code. If time ordering is critical, can the ordering be correctly reconstructed later?

- Third, verify that moving synchronization to another place in the code will not cause similar performance problems in the new location. One way to do this is to show that the number of synchronizations will decrease dramatically because of your work (such as in the event-counting example above).

**Additional Resources**

[Intel® Software Network Parallel Programming Community](#)

[OpenMP® Specifications](#)

