Use Synchronization Routines Provided by the Threading API Rather than Hand-Coded Synchronization

<keywords: threading, synchronization, spin-wait, Hyper-Threading, Win32 threads, OpenMP, Pthreads>

Abstract

Application programmers sometimes write hand-coded synchronization routines rather than using constructs provided by a threading API in order to reduce synchronization overhead or provide different functionality than existing constructs offer. Unfortunately, using hand-coded synchronization routines may have a negative impact on performance, performance tuning, or debugging of multi-threaded applications.

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

It is often tempting to write hand-coded synchronization to avoid the overhead sometimes associated with the synchronization routines from the threading API. Another reason programmers write their own synchronization routines is that those provided by the threading API do not exactly match the desired functionality. Unfortunately, there are serious disadvantages to hand-coded synchronization compared to using the threading API routines.

One disadvantage of writing hand-coded synchronization is that it is difficult to guarantee good performance across different hardware architectures and operating systems. The following example is a hand-coded spin lock written in C that will help illustrate these problems:

```
#include <ia64intrin.h>

void acquire_lock (int *lock)
{
  while (_InterlockedCompareExchange (lock, TRUE, FALSE) == TRUE);
}

void release_lock (int *lock)
{
  *lock = FALSE;
}
```

The `_InterlockedCompareExchange` compiler intrinsic is an interlocked memory operation that guarantees no other thread can modify the specified memory location during its execution. It first compares the memory contents of the address in the first argument with the value in the third argument, and if a match occurs, stores the value in the second argument to the memory address specified in the first argument. The original value found in the memory contents of the specified address is returned by the intrinsic. In this example, the `acquire_lock` routine spins until the contents of the memory location `lock` are in the unlocked state (`FALSE`) at which time the lock is acquired (by setting the contents of `lock` to `TRUE`) and the routine returns. The `release_lock` routine sets the contents of the memory location `lock` back to `FALSE` to release the lock.

Although this lock implementation may appear simple and reasonably efficient at first glance, it has several problems:

- If many threads are spinning on the same memory location, cache invalidations and memory traffic can become excessive at the point when the lock is released, resulting in poor scalability as the number of threads increases.
- This code uses an atomic memory primitive that may not be available on all processor architectures, limiting portability.
- The tight spin loop may result in poor performance for certain processor architecture features, such as Intel® Hyper-Threading Technology.
- The `while` loop appears to the operating system to be doing useful computation, which could negatively impact the fairness of operating system scheduling.

Although techniques exist for solving all these problems, they often complicate the code enormously, making it difficult to verify correctness. Also, tuning the code while maintaining portability can be difficult. These problems are better left to the authors of the threading API, who have more time to spend verifying and tuning the synchronization constructs to be portable and scalable.

Another serious disadvantage of hand-coded synchronization is that it often decreases the accuracy of programming tools for threaded environments. For example, the Intel® Parallel Studio tools need to be able to identify synchronization constructs in order to provide accurate information about performance (using Intel® Parallel Amplifier) and correctness (using Intel® Parallel Inspector) of the threaded application program.

Threading tools are often designed to identify and characterize the functionality of the synchronization constructs provided by the supported threading API(s). Synchronization is difficult for the tools to identify and understand if standard synchronization APIs are not used to implement it, which is the case in the example above.

Sometimes tools support hints from the programmer in the form of tool-specific directives, pragmas, or API calls to identify and characterize hand-coded synchronization. Such hints, even if they are supported by a particular tool, may result in less accurate analysis of the application program than if threading API synchronization were used: the reasons for performance problems may be difficult to detect or threading correctness tools may report spurious race conditions or missing synchronization.

Advice

Avoid the use of hand-coded synchronization if possible. Instead, use the routines provided by your preferred threading API, such as a `queuing_mutex` or `spin_mutex` for Intel® Threading Building Blocks, `omp_set_lock/omp_unset_lock` or `critical/end critical` directives for OpenMP*, or `pthread_mutex_lock/pthread_mutex_unlock` for Pthreads*. Study the threading API synchronization routines and constructs to find one that is appropriate for your application.

If a synchronization construct is not available that provides the needed functionality in the threading API, consider using a different algorithm for the program that requires less or different synchronization. Furthermore, expert programmers could build a custom synchronization construct from simpler synchronization API constructs instead of starting from scratch. If hand-coded synchronization must be used for performance reasons, consider using pre-processing directives to enable easy replacement of the hand-coded synchronization with a functionally

equivalent synchronization from the threading API, thus increasing the accuracy of the threading tools.

Usage Guidelines

Programmers who build custom synchronization constructs from simpler synchronization API constructs should avoid using spin loops on shared locations to avoid non-scalable performance. If the code must also be portable, avoiding the use of atomic memory primitives is also advisable. The accuracy of threading performance and correctness tools may suffer because the tools may not be able to deduce the functionality of the custom synchronization construct, even though the simpler synchronization constructs from which it is built may be correctly identified.

Additional Resources

Intel® Software Network Parallel Programming Community

John Mellor-Crummey, "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors." *ACM Transactions on Computer Systems*, Vol. 9, No. 1, February 1991. Pages 21-65.

Intel Pentium 4 and Intel Xeon Processor Optimization Reference Manual, Chapter 7: "Multiprocessor and Hyper-Threading Technology." Order Number: 248966-007.

Intel Parallel Studio

OpenMP* Specifications

Intel® Threading Building Blocks

Intel Threading Building Blocks for Open Source

James Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, Inc. Sebastopol, CA, 2007.