Using Tasks Instead of Threads

Abstract

Tasks are a light-weight alternative to threads that provide faster startup and shutdown times, better load balancing, an efficient use of available resources, and a higher level of abstraction. Three programming models that include task based programming are Intel® Threading Building Blocks (Intel® TBB) and OpenMP®. This article provides a brief overview of task-based programming and some important guidelines for deciding when to use threads and when to use tasks.

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

Programming directly with a native threading package is often a poor choice for multithreaded programming. Threads created with these packages are logical threads that are mapped by the operating system onto the physical threads of the hardware. Creating too few logical threads will undersubscribe the system, wasting some of the available hardware resources. Creating too many logical threads will oversubscribe the system, causing the operating system to incur considerable overhead as it must time-slice access to the hardware resources. By using native threads directly, the developer becomes responsible for matching the parallelism available in the application with the resources available in the hardware.

One common way to perform this difficult balancing act is to create a pool of threads that are used across the lifetime of an application. Typically, one logical thread is created per physical thread. The application then dynamically schedules computations on to threads in the thread pool. Using a thread pool not only helps to match parallelism to hardware resources but also avoids the overheads incurred by repeated thread creation and destruction.

Some parallel programming models, such as Intel TBB and the OpenMP API provide developers with the benefits of thread pools without the burden of explicitly managing the pools. Using these models, developers express the logical parallelism in their applications with tasks, and the runtime library schedules these tasks on to its internal pool of worker threads. Using tasks, developers can focus on the logical parallelism in their application without worrying about managing the parallelism. Also, since tasks are much lighter weight than threads, it is possible to express parallelism at a much finer granularity.

A simple example of using tasks is shown below. The function fibTBB calculates the n\textsuperscript{th} Fibonacci number using a TBB task_group. At each call where n >= 10, a task group is created and two tasks are run. In this example, a lambda expression (a feature in the proposed C++0x standard) that describes each task is passed to the function run. These calls spawn the tasks, which makes them available for threads in the thread pool to execute. The subsequent call to the function wait blocks until all of the tasks in the task group have run to completion.

```c
int fibTBB(int n) {
    if( n<10 ) {
        return fibSerial(n);
    } else {
        int x, y;
```
```cpp
tbb::task_group g;
g.run([&]{x=Fib(n-1);}); // spawn a task
g.run([&]{y=Fib(n-2);}); // spawn another task
g.wait(); // wait for both tasks to complete
return x+y;
}
```

The routine `fibSerial` is presumed to be a serial variant. Though tasks enable finer grain parallelism than tasks, they still have significant overhead compared to a subroutine call. Therefore, it generally pays to solve small subproblems serially.

Other libraries that support tasks include the OpenMP API. Unlike Intel TBB, both of these models use compiler support, which makes their interfaces simpler but less portable. For example, the same Fibonacci example shown above using TBB tasks is implemented as `fibOpenMP` below using OpenMP tasks. Because OpenMP requires compiler support, simpler pragmas can be used to denote tasks. However, only compilers that support the OpenMP API will understand these pragmas.

```cpp
int fibOpenMP( int n ) {
    int i, j;
    if( n < 10 ) {
        return fibSerial(n);
    } else {
        // spawn a task
        #pragma omp task shared( i ), untied
        i = fib( n - 1 );
        // spawn another task
        #pragma omp task shared( j ), untied
        j = fib( n - 2 );
        // wait for both tasks to complete
        #pragma omp taskwait
        return i + j;
    }
}
```

Intel TBB and the OpenMP API manage task scheduling through work stealing. In work stealing, each thread in the thread pool maintains a local task pool that is organized as a deque (double-ended queue). A thread uses its own task pool like a stack, pushing new tasks that it spawns onto the top of this stack. When a thread finishes executing a task, it first tries to pop a task from the top of its local stack. The task on the top of the stack is the newest and therefore most likely to access data that is hot in its data cache. If there are no tasks in its local task pool, however, it attempts to steal work from another thread (the victim). When stealing, a thread uses the victim’s deque like a queue so that it steals the oldest task from the victim’s deque. For recursive algorithms, these older tasks are nodes that are high in the task tree and therefore are large chunks of work, often work that is not hot in the victim’s data cache. Therefore, work stealing is an effective mechanism for balancing load while maintaining cache locality.

The thread pool and the work-stealing scheduler that distributes work across the threads are hidden from developers when a tasking library is used. Therefore, tasks provide a high-level abstraction that lets users think about the logical parallelism in their application without worrying about managing the parallelism. The load balancing provided by work-stealing and the low creation and destruction costs for tasks make task-based parallelism an effective solution for most applications.
Usage Guidelines

While using tasks is usually the best approach to adding threading for performance, there are cases when using tasks is not appropriate. The task schedulers used by Intel TBB and the OpenMP API are non-preemptive. Tasks are therefore intended for high-performance algorithms that are non-blocking. They still work well if the tasks rarely block. However, if tasks block frequently, there is a performance loss because while a task is blocked, the thread it has been assigned to cannot work on any other tasks. Blocking typically occurs while waiting for I/O or mutexes for long periods. If threads hold mutexes for long periods, the code is not likely to perform well, regardless of how many threads it has. For blocking tasks, it is best to use threads rather than tasks.

Even in cases when using tasks is best, sometimes it’s not necessary to implement the tasking pattern from scratch. The Intel TBB library provides not only a task interface but also high-level algorithms that implement some of the most common task patterns, such as `parallel_invoke`, `parallel_for`, `parallel_reduce` and `pipeline`. The OpenMP API also offers parallel loops. Since these patterns have been tuned and tested, it’s best to use these high-level algorithms whenever possible.

The example below shows a simple serial loop and a parallel version of the loop that uses the `tbb::parallel_for` algorithm:

```cpp
// serial loop
for (int i = 0; i < 10000; ++i)
  a[i] = f(i) + g(i);

// parallel loop
tbb::parallel_for( 0, 10000, [a](int i) { a[i] = f(i) + g(i); } );
```

In the example above, the TBB `parallel_for` creates tasks that apply the loop body, in this case `a[i] = f(i) + g(i)`, to each of the elements in the range `[0,10000)`. The `&` in the lambda expression indicates that variable `a` should be captured by reference. When using a `parallel_for`, the TBB runtime library chooses an appropriate number of iterations to group together in a single task to minimize overheads while providing ample tasks for load balancing.

Additional Resources

Intel® Software Network Parallel Programming Community

Intel® Threading Building Blocks

Intel Threading Building Blocks for Open Source


OpenMP* Specifications