Putting Intel® Threading Building Blocks to Work

Nicolae Popovici
Intel GmbH
Dornacherstr. 1
85622 Feldkirchen
+49 174 2191971
Nicolae.o.popovici@intel.com

Thomas Willhalm
Intel GmbH
Dornacherstr. 1
85622 Feldkirchen
+49 174 2191966
Thomas.willhalm@intel.com

ABSTRACT

Intel® Threading Building Blocks (TBB) was designed to simplify programming for multi-core platforms. By introducing a new way of expressing parallelism, developers can focus on efficient scalable parallel program design and avoid dealing with low level details of threading. This talk will focus on the task-based approach to threading of the Intel® TBB. We will show step by step what needs to be done in order to take advantage of the TBB task based programming model in your code.

Categories and Subject Descriptors

D.1.3 Parallel programming

General Terms

Performance, Design, Languages

Keywords

Multi-Core, C++, tasked-based programming

1. Introduction

Today, the majority of Intel platforms are based on multi-core technology. To exploit the performance potential of multi-core processors, applications must be threaded. There is set of threading strategies that you can use to thread your application. Modern operating systems provide APIs to create and manage threads. These APIs are very flexible, but they also are very low level. Furthermore, an OS thread-based solution is not portable. You can use threaded libraries to develop cross-platform solutions, but only if the libraries are not too specialized. Alternatively, there are threading libraries that provide cross-platform wrappers for operating systems threading API. They provide cross-platform compatibly and the flexibility to express a wide range of threading models. However, a developer still has to rewrite common parallel algorithms and data structures such as thread pools, thread-safe container classes, or fast synchronization primitives. Last but not least, it is very hard to avoid thread safety issues when using threads.

Concerning performance, there are a number of issues that are difficult to avoid when using threads. The most prominent one is the CPU's under- or oversubscription. It occurs when multiple software components create threads that compete for CPU, and the number of active threads is higher than the number of available cores. To avoid this problem, programmers must develop an algorithm and data structures to manage the level of CPU utilization across all software components. Often, it is hard to get it right and requires expert level of performance tuning.

OS threads are managed by the OS thread scheduler. Schedulers typically still distribute time slices in a round-robin fashion, so each thread gets its fair share of time to execute on CPU. Fair scheduling, however, does not consider shared caches or NUMA. A threading API is too low-level, and it typically provides only crude methods to balance the load. This often may lead to the under-utilization of CPUs and poor scalability when there are not enough running threads to keep the CPUs busy. TBB is a threading abstraction library. It relies on generic programming, which means that TBB interfaces do not limit their application to particular types which enables broad applicability of the best algorithms with the fewest constraints. Generic programming enables Intel Threading Building Blocks to be flexible and work sufficiently with custom, user-defined types.

The Threading Building Blocks [1,2,3] provide a portable high-level generic implementation of parallel algorithms and concurrent data structures in C++. The library automatically creates and manages a thread pool, and it does dynamic load balancing of parallel work. In order to achieve this, the TBB library introduces a concept of logical tasks that allows you to abstract from the low-level threading details, which are already implement correctly. You specify tasks that you want to be executed concurrently and the TBB task scheduler will map these tasks onto threads in an efficient manner. Furthermore, TBB fully supports nested and recursive parallelism. It is therefore designed to provide scalable performance for the computationally intense portions of applications.

The Intel® Threading Building Blocks are currently available on Linux, Windows, and Mac OS. The library has been open-sourced and is currently ported to further platforms [4].

2. Generic Parallel Algorithms and Data Structures

Parallel generic algorithms and concurrent containers are high-level utilities for parallel programming provided by the Threading Building Blocks. Parallel algorithms provide highly tuned implementation of commonly used parallel patterns such as loop parallelization, a data flow pipeline, or parallel sort. Their implementation relies on the generic features of C++ like templates classes and function-like objects and follows the spirit of the Standard Template Library.

Copyright is held by the author/owner(s).
IWMSE'08, May 11, 2008, Leipzig, Germany.
ACM 978-1-60558-031-9/08/05.

The concurrent containers in TBB - concurrent hash map, vector, and queue - are thread-safe data containers that are based on fine grain locking and lockless algorithms. The lowest library level is synchronization primitives and memory allocators.

The middle layer is a task scheduler. It is an engine that drives parallel generic algorithms and hides the complexity of thread management. It provides C++ interface that you can use to create your own parallel algorithms if the packaged parallel algorithms do not meet your requirements.

3. Task-Based Programming

The core of the Intel Threading Building Blocks library is the task scheduler. It is designed to address common performance issues of parallel programming with operating systems threads. The task scheduler manages the internal thread pool and it was designed to hide the complexity of operating systems threads. Task scheduler manages logical tasks that the developer creates, and maps them onto OS threads.

For computations that do not wait for external devices, highest efficiency usually occurs when there is exactly one running thread per core. Otherwise, there can be inefficiencies from mismatch, i.e. over- or under-subscription. The task scheduler therefore starts only one thread per core and maps logical tasks onto threads in a way that tolerates interference by other threads from the same or other processes. Unlike an OS thread scheduler; the TBB task scheduler is not fair and non-preemptive. Since you describe parallelism in terms of logical tasks, the task scheduler has some information about high-level organization of the program, and therefore can trade fairness for efficiency.

One of the key advantages of a logical task is that it is much lighter than a thread. A task is a C++ object while an OS thread is a kernel object which has its own copy of a lot of resources such as register state and stack. On Linux systems, starting and terminating a task is around 18 times faster than starting and terminating a thread. On Windows systems, this ratio is more than a hundred. Using tasks therefore dramatically decreases the parallel overhead. The task scheduler does load balancing, i.e., in addition to use the right number of threads, the task scheduler tries to distribute the total work evenly across all threads by using a work-stealing algorithm.

To use the task based programming in TBB, a developer will need to derive its own task class from the TBB abstract base task class. The base task class has a pure virtual method `execute` that must be implemented for a new logical task. Starting from a root task, logical tasks will be created, which again spawn other logical task. This results in a tree of tasks, which is traversed by the task scheduler.

The library puts at our disposal a number of methods that we can use for task allocation. A parent task uses a reference counter to know how many child tasks need to complete. After allocating a task, the developer will need to inform the task scheduler that a new created task is ready to be picked up for execution. In the Threading Building Blocks library, this can be achieved through different flavors of the spawn method. Spawning a task does not mean creating a new thread but just put the task in the ready task queue that each thread can then access. Waiting for the newly created children tasks to complete is accomplished by a call to `wait_for_all()` function. An important difference to note is that this call will not block the calling thread like any OS wait-function would do.

Let us now have a look at the steps that need to be performed within the main function for a task-based program. To use the task scheduler within TBB, one needs to initialize it first by instantiating a `task_scheduler_init` object. By default, the constructor of this object immediately creates a thread pool and initializes the internal data structures of the task scheduler. The most important data structure is the per-thread task. After the allocation of memory for your root task (using the static method `allocate_root)` and setting its reference counter to 1, you can spawn the root task. The task scheduler will place this new task to the task deque and the thread associated with this deque will start executing the body of the method `execute` of the scheduled task.

When one of the TBB working threads executes the body of this root task, the root task actually performs a for-loop where it allocates memory for a new task and spawns it. Each spawned task is added to one of the task pools, where the TBB worker threads pull tasks from and runs their method `execute`. When a TBB worker thread completes all the tasks from its task pool, it starts stealing tasks randomly from other task pools. This work-stealing technique is an easy but efficient method to balance the load in parallel execution. More details than this oversimplified description of how the TBB task scheduler works can be found in [1].

4. Summary

We have seen that task-based programming is an easy and efficient method to implement computational-intensive applications on multi-core systems. The Intel® Threading Building Blocks provide an easy but powerful interface that allows a natural implementation using tasks in C++ without re-implementing common parallel algorithms and concurrent data structures. The core of TBB is its task scheduler, which provides an effective means to efficiently create and distribute tasks.

5. REFERENCES

[1] Reinders, J., 2007. Intel Threading Building Blocks. O'Reilly

[2] Crownie, J.. 2007. Building High Performance Threaded Applications using Libraries. Presented at Parallel Architectures and Compilation Techniques (PACT) 2007. http://pact07.cs.tamu.edu/TBB_PACT2007.pdf

[3] Deilmann, M. and Willhalm T., 2007. Intel® Threading Building Blocks – ein templatebasiertes Programmiermodell für moderne Mehrkernarchitekturen. In LinuxMagazin 05/2007.

[4] Intel® Threading Building Blocks 2.0 for Open Source http://threadingbuildingblocks.org/