TBB initialization, termination, and resource management details, juicy and gory.

By Andrey Marochko (Intel) (14 posts) on April 9, 2011 at 8:35 am

Well, maybe more essential than juicy, and rather treacherous than gory. As I noted in my previous blog introducing a major task scheduler extension – support for task and task group priorities, TBB has been steadily evolving ever since its inception. My recent interactions with a few teams using TBB both inside and outside of Intel made me realize that simplicity of the task scheduler life cycle and configuration management peculiar to its early versions has faded into oblivion, morphed by a series of fixes addressing various corner cases and a barrage of changes improving library usability in general and composability of TBB based components in particular. Besides, as it often happens, fixing some issues brings in other ones instead, even if not as nasty as the original ones…

Though at least some of the modifications have already been mentioned in various places, even if only shortly or indirectly, there is no single document describing the topic. With all this not making the life of TBB users easier, I've decided to continue the line of my other recent blog describing earlier undocumented changes in TBB scheduler, and write an overview of the current state of the matter related to its initialization, termination, and configuring. More specifically we will talk about *tbb::task_scheduler_init*, concurrency level control, lazy workers creation, and about auto initialization.

To facilitate the discussion I'll be referring to the following diagram that summarizes major internal components constituting TBB task scheduler.


Main TBB scheduler components

If you want some more background information why TBB arrived to a design depicted on the scheme, and have not yet read the composability blog I mentioned in the beginning, you could have a look at it now :).

Initialization phase.

After TBB dynamic library is loaded into an application and its static initialization is completed, none of the components mentioned on the picture exist yet (except of course one or more application threads).

When an application thread creates the first (in this thread) instance of tbb::task_scheduler_init object, it becomes a master thread (as long as TBB is concerned), and the following chain of events takes place.

- If this *task_scheduler_init* instance is the first one not only in this thread but also in the whole process, then
  - **RML** (Resource Management Layer) singleton is created. This is a component that hosts a pool of TBB worker threads. It comes in two incarnations, a private one that is built in into TBB library, and a shared RML that is supplied as a standalone DLL. They differ a little in their behavior, but as normally the private one is used, the subsequent description will assume its policies.Note that no actual threads are created at this moment.
  - **Market** singleton is instantiated. This component is responsible for assigning worker threads to different arenas.
- **Task dispatcher** associated with this master thread is created. Any thread executing TBB tasks (whether master or worker) has its own instance of this component, which is stored in a TLS slot. [BTW, if you ever look into TBB sources, task dispatcher is implemented in the form of *generic_scheduler* <- *custom_scheduler* class hierarchy.]
- **Arena** associated with this master thread is allocated. Just created local task dispatcher registers itself in its arena, arena is added into market's list that tracks all existing arenas, and market refcount is incremented.

An important thing to keep in mind is that during this process essential limits are established. The first one is the number of arena slots that define maximal number of workers available for parallel algorithms started by this master (or, in other words, their maximal concurrency level).

The second limit is established when the market is created, and sets the ceiling on the total number of workers available to all master threads. It is determined as one less than the greater of the following two values: amount of threads specified by the argument of *task_scheduler_init* constructor, and current hardware concurrency (i.e. amount of logical CPUs visible to OS).

For the sake of illustration consider the following example. An application is started on an 8 core machine and one of its threads (T1) creates *task_scheduler_init* object specifying 4 as its argument. As the result T1 will get arena with 3 slots for worker threads, and since it is the first thread to initialize TBB scheduler in the process a market instance will be created with upper limit of workers set to 7 (as the application is running on the 8 core machine). If afterwards other threads (T2 and T3) initialize scheduler requesting 8 and 16 threads, their arenas will get 7 slots each. T3 does not end up with 15 slots as the market has already been initialized and is limited with a total of 7 workers only.

request specifies different concurrency level? For example T1 repeatedly requests 8 threads (that is 7 workers), and T3 asks for 6 threads (= 5 workers). Whether for good or for bad, in both cases absolutely nothing happens, except for a refcount on the local task dispatcher being incremented.

Thus with the current TBB version it is impossible to change concurrency level of parallel algorithms started from a thread after this thread has initialized the scheduler. Yet, if all *task_scheduler_init* instances created by a thread are destroyed, then new *task_scheduler_init* object can set different concurrency level for this thread, though not without more caveats, which we'll come to in a few moments, when we start discussing deinitialization process.

Execution phase.

Lazy thread creation

Now that we've finished talking about the first phase of the initialization process, let's see what happens next. As I already noted above, when RML server (thread pool manager) is instantiated, no OS threads are created. Threads creation is postponed until the first task is spawned or enqueued (in particular this happens when a TBB parallel algorithm is invoked). In this case RML creates exactly as many threads as the active arena needs.

Returning to our example, if T1 (that specified its interest in 3 workers) was the first to start parallel work, RML will create 3 worker threads, despite its capacity of 7 workers. If then, while T1 is executing its parallel work, another thread T4 with requested concurrency level of 6 kicks in, RML will create remaining 4 workers (bringing their total to its limit of 7), in attempt to satisfy cumulative request from active arenas.

If, however, T4 starts its parallel work after T1 finished its, then RML will additionally create only 2 workers, as the pending cumulative request is that of T4, and it is only 5 workers. This way TBB caps system resources consumption by allocating only minimally necessary amount of threads.

Note that lazy thread creation was first introduced in TBB 3.0 Update 1. Before then the whole thread pool was allocated at the moment of RML server instantiation.

Workers distribution

All right, we've already made that far, and there's just one more situation possible during execution phase that remains unvivisected. What happens when there are several master threads concurrently executing parallel algorithms and their cumulative demand for workers exceeds market capacity? Actually it's pretty easy. The market will allot worker quotas proportionally to each master's request so that its limit is not topped (with possible fluctuations in the distribution caused by integer arithmetic).

Continuing our example, let T1, T2, and T4 are those concurrent masters. Their total demand is 3 + 5 + 7 = 15 workers. With market's limit of 7, allotment for T1 will be 3/15*7 = 1, for T2 – 5 / 15 * 7 = 2, and T3 will get remaining 4 workers. Depending on the order of registration in the market, the extra 1 (accumulated remainder after rounding down) may go to any of these arenas (in our example I assumed that arenas registered in the order of their thread indices, and so it went to the fattest one, bringing its quota from 3 to 4).

It is also important to realize, that there will always be some timeout between the moment the market updates worker quotas, and the moment when workers actually migrate between arenas. This happens because a worker can discover that it needs to leave its current arena in order to join another one only when it is both in its stealing loop (that is does not have tasks in its local task pool) and does not execute a nested parallel algorithm. Thus re-establishing fairness of worker threads distribution may take a noticeable time in some cases.

Termination phase.

When the last *task_scheduler_init* object created by this thread is destroyed, the actions opposite to initialization events take place, though not exactly in the mirror fashion. First, the local task dispatcher object is destroyed. During its destruction sequence it notifies its arena that the master thread is leaving. If the arena does not contain any tasks and there are no workers attached, it is destroyed as well.

It is possible, however, that at the moment of the local dispatcher destruction its arena either contains tasks, or has workers still attached to it (or both). E.g. tasks scheduled via *task::enqueue()* method, can be used in a fire-and-forget manner. That is master thread that enqueued them, is allowed to not wait for their completion. Thus it is completely well-formed usage when an application thread initializes TBB task scheduler, allocates a task, enqueues it, and immediately terminates the scheduler.

The second kind of situation is possible because of the following scheduler dispatch loop peculiarity. When you run a parallel algorithm (either a predefined one, like *tbb::parallel_for()*, or by spawning a task and calling *task::wait_for_all()*) TBB guarantees that by the moment control returns to the caller all tasks that constituted this parallel computation are executed. They are also already destroyed, if only you do not recycle some of them manually. This, however, does not mean that worker threads have already left the arena. In fact, because of distributed and loosely coupled nature of TBB scheduler design (necessary to ensure high scalability) they may not even know yet that there is no more work in the arena. A worker will repeat random stealing attempts during a short time after it finished its last locally available task before it undertakes exhaustive (and rather costly) arena inspection, if only some other worker that finished its last task earlier has not already done such inspection and found arena empty. All in all, as the result arena may still contain some workers when all its task are done and master thread leaves.

In both cases master thread obviously cannot destroy its arena, and so it leaves it alive until all the remaining tasks task are executed, and then the last leaving worker will destroy it.

As part of its destruction sequence, arena detaches itself from the market, and decrements market's refcount. If market's refcount drops to zero, it is also destroyed, along with resource (thread pool) manager.

Completing our discussion of the task scheduler deinitialization, I must mention a very unpleasant effect (I guess one could call it a bug) that may happen because workers can remain in their arenas some time after all the work is done. Normally you do not see this problem, but if the application exits or the module that used TBB is unloaded immediately after the last parallel algorithm completion, it may, well, crash…

Definitely a bug :(… This happens because in such cases TBB dynamic library may become unloaded from the process address space while worker threads are still executing its code and reference its data. I won't be making excuses, just refer those who wants to learn a little more about why unloading dynamic libraries in multithreaded apps is pain in the ass to this Microsoft's whitepaper, and promise that this bug will be fixed in one of the next TBB releases.

Auto-initialization.

So far we only considered scenarios involving explicit usage of *tbb::task_scheduler_init*. Before TBB 2.2 this was the sole (and therefore obligatory) way of TBB task scheduler initialization. Starting with TBB 2.2 any operation with tasks (allocation, spawning, enqueuing, and destruction) will initialize task scheduler automatically, if this has not been done in this thread yet. Effects of automatic initialization are mostly identical to explicit creation of *task_scheduler_init* object. There are only two peculiarities that may affect behavior of your program.

The first one is lack of ability to control concurrency level of the auto-initialized task dispatcher, which is set to the current hardware concurrency (amount of logical CPUs visible to OS). Strictly speaking, since TBB 3.0 Update 4 you can change this default by modifying process affinity mask. See this blog for more details. But this workaround in its turn has substantial limitations. First, changing hardware affinity is a costly operation. Second, in the current implementation, TBB performs analysis of the process affinity mask only once, and caches the detected value to be used as the default concurrency level from then on.

As the default concurrency level (which allows for full hardware resources utilization) is normally what you want anyway, this auto-initialization deficiency is rarely an issue. Yet, in combination with the second peculiarity of auto-initialization they may become a bigger problem.

As I noted above, when *task_scheduler_init* is used explicitly, it is possible to re-initialize task scheduler in the given thread with another concurrency level. Moreover, if you have complete control over the whole application, and manage to terminate task scheduler in all threads, you can even create new market with another global limit on the number of workers. Though, honestly this is a treacherous trick to rely on, because of workers lingering in arenas.

In case of auto-initialization, local task dispatcher will be destroyed only when its thread exits. Often this happens only when the application completes. This means that auto-initialization, with all its convenience, may restrict your ability to control concurrency of parallel computations. The only consolation is that fortunately this is rarely an issue in practice.

Conclusion.

Wow, you did it! Hopefully reading through this not the shortest possible post helped you to understand what's going on under the hood of TBB a little better, and this better understanding will help you some time to write a better application.

At last, before saying good bye, I'd like to highlight the fact that this blog describes mechanics of TBB as it is in its version 3.0 Update 6. Since you've certainly noticed that the current design imposes some restrictions that have potential of becoming burdensome in some situations (even if only infrequently), there will surely be changes in the future releases improving TBB scheduler configurability. So look out!

Categories: Open Source, Parallel Programming, Software Tools
Tags: Composer, concurrency, multithreading, parallel studio, parallelism, scheduler, task scheduling, TBB, Threading Building Blocks

Comments (0)

Trackbacks (1)

- TBB initialization, termination, and resource management details
  April 13, 2011 2:03 PM PDT

Leave a comment

Email (required, will not be displayed on this page)

Your URL (optional)

Comment*

submit