Intel Thread Building Blocks, Part IV

SPD course 2018-19
Massimo Coppola
16/05/2019

Mutexes

- TBB Classes to build *mutex lock objects*
- The lock object will
  - Lock the associated data object (the mutex) for use by the current thread
  - Allow any thread to wait and obtain the lock according to a specific semantics for locking
  - Have a scope locking pattern
    - TBB releases locks when destroyed at end of scope
    - Automatically release locks when they are no longer in scope, including the case of uncaught exceptions
      - No need for std::lock_guard like in C++11/14
  - Possibly account for reading and writing behaviour of the locking thread

Mutexes are low-level

- TBB mutexes and locks are designed to provide best performance and lowest overhead in different situations
  - low/high contention, small/large critical sections…
- It is up to the programmer to avoid
  - deadlocks (when threads get more lock….)
  - performance degradation due to SW lockout
  - performance loss due to thread de-scheduling while inside critical sections
- C++ style mutexes are avoided in TBB because they are not exception safe

Mutex and scoped lock

- Mutex and lock; example of scoped locking
  - The mutex and its locks are not copiable nor moveable

```
int count;
tbb::mutex countMutex;

{   // Implements ANNOTATE_LOCK_ACQUIRE()
    tbb::mutex::scoped_lock lock(countMutex);
    result = count++;
    // Implicit ANNOTATE_LOCK_RELEASE() when leaving the scope below.
}  // scoped lock is automatically released here
```

- Mutexes / Scoped_lock basic primitives
  - Type signatures
  - Construct
  - Construct and acquire
  - Destroy (and possibly release)
  - acquire
  - Try_acquire
  - release

Pseudo-Signature	Semantics
`M()`	Construct unlocked mutex.
`~M()`	Destroy unlocked mutex.
`typename M::scoped_lock`	Corresponding scoped-lock type.
`M::scoped_lock()`	Construct lock without acquiring mutex.
`M::scoped_lock(M&)`	Construct lock and acquire lock on mutex.
`M::~scoped_lock()`	Release lock (if acquired).
`M::scoped_lock::acquire(M&)`	Acquire lock on mutex.
`bool M::scoped_lock::try_acquire(M&)`	Try to acquire lock on mutex. Return true if lock acquired, false otherwise.
`M::scoped_lock::release()`	Release lock.
`static const bool M::is_rw_mutex`	True if mutex is reader-writer mutex; false otherwise.
`static const bool M::is_recursive_mutex`	True if mutex is recursive mutex; false otherwise.
`static const bool M::is_fair_mutex`	True if mutex is fair; false otherwise.

Mutexes

- Mutexes are available in different implementations, with various features
  - Scalability – whether lock may withstand heavy contention with low overhead
  - Fairness – whether lock takes into account the order of lock attempts and prevents any starvation
  - Reentrant behaviour – wheter recursive locking is allowed and correctly managed (no undue overhead, no misbehaving / deadlock)
  - Yield / Block – whether the thread waiting for a lock may be suspended and yield the CPU core
  - Size – size of the lock structure, relevant when a large number of mutexes are used to fine-grain lock portions of dynamic data structures

Mutex and recursive mutex

- Plain mutex is a wrapper class for the native mutex of the OS
  - i.e. pthreads, except for Windows
  - They add the scope-locking behaviour on top
  - Other behavior depends on the OS
- recursive_mutex can be acquired repeatedly by the same thread
  - Allow easier use with some recursive code
  - Performs proper lock counting
  - A (different) wrapper around the OS mutexes

Spin_mutex, Queueing_mutex

Spin_mutex = simplest lock implementation

- Spinning = busy-waiting checking for the lock to become available
- Spin locks are not fair or efficient
- Good for very short, quickly executed scopes
  - Avoid it with very high contention

Queueing_mutex is the swiss' army knife

- Queue locks provide fairness by managing the waiting threads as a FIFO queue
- Implementation is scalable and moderate overhead

Transactional memory support

- Groups several read and writes from memory made within a region of program code (the lock scope) into a single *atomic transaction*
- HW support detects if the read and write regions of a transaction are touched by any other thread (Bernstein conditions check)
- All the writes are only performed if there was no concurrent conflict
- Otherwise HW performs a full rollback of the processor status and caches
  – thread restarts before the lock attempt

Speculative spin mutex

- Exploits HW support of transactional memory
  - e.g. TSX machine instructions if available
  - degrades to a spinning lock if no HW support
- No actual locking, no spinning (just marks the start of an "atomic transaction" for the CPU)
- Convenient when conflict is unlikely and critical section is short
  - No overhead if no conflict
  - Penalty in case of rollback is the whole critical section
- Comes also in a reader/writer variant
- Subject to HW limitations, typically
  - Granularity of memory access is a cache line
  - All read ad write regions must fit in L1 cache

Read write lock concept

- Models the reader-writer problem
- Has a read and a write mode of acquiring the lock
  - Multiple readers or (XOR) at most one writer are allowed to hold the lock
  - Holder can upgrade the lock (reader to writer)
    - possibly with an additional wait
    - possibly releasing and reacquiring the lock
  - Holder can also downgrade (writer to reader)
    - possibly allowing more readers in
- Some TBB locks currently have a r/w version
  - Spin, speculative spin, queueing
  - RW locks are not recursive

Null_mutex and null_rw_mutex

- Do not perform any actual locking
- No added size for the lock
- Rationale:
  – templates classes for concurrent data structures
  – use a mutex class as a parameter for flexibility
  – Allow reusing the template also for lock-free versions of the structures

Summary of mutexes

	Scalable	Fair	Reentrant	Long Wait	Size
`mutex`	OS dependent	OS dependent	No	Blocks	>=3 words
`recursive_mutex`	OS dependent	OS dependent	Yes	Blocks	>=3 words
`spin_mutex`	No	No	No	Yields	1 byte
`speculative_spin_mutex`	HW dependent	No	No	Yields	2 cache lines
`queuing_mutex`	Yes	Yes	No	Yields	1 word
`spin_rw_mutex`	No	No	No	Yields	1 word
`speculative_spin_rw_mutex`	HW dependent	No	No	Yields	3 cache lines
`queuing_rw_mutex`	Yes	Yes	No	Yields	1 word
`null_mutex`	-	Yes	Yes	-	empty
`null_rw_mutex`	-	Yes	Yes	-	empty

Task management

- The task scheduler manages the computation of a set of tasks by a pool of worker threads
- Tasks are connected in a tree
  - From the initial task more are dynamically spawn, and are distributed to worker threads
  - In general we can have a forest (a set of disjoint trees)
- Each worker thread has a dequeue of tasks
  - The dequeue is ordered by task oldness (older tasks up, newer down)
  - Push and pop of new tasks to compute normally happens at the bottom
  - Favors smaller tasks, depth-first expansion, helps reducing the stack occupation

Task management and scheduling

- Worker thread prefer operating on their local task dequeue
  - Lower overhead and less contention on runtime structures
- When a dequeue is empty, the thread performs work stealing from a random thread
  - Stealing happens at the top of the dequeue = older tasks with more potentail for further parallelism – breadth first expansion of the task tree

Task and task management

- Task class is mainly for implementing new algorithms
  - Provides an execute() method that is called by a worker thread
  - Execute can also provide hints about task affinity to the scheduler
- Tasks can be spawn explicitly
  - Doing so naively can result in poor performance
  - Tasks can spawn child tasks (more parallelism) as well as continuation tasks
  - Computed tasks can be recycled
    - turn into a different tasks and enqueued again
    - avoid allocation overhead
  - More complex features (e.g. scheduler bypass)
- empty task = do nothing

Scheduler initialization

- Task_scheduler_init provides means for the user to customize the scheduler
  - When the scheduler is constructed/destroyed
  - How many worker threads the scheduler uses
  - The stack size of worker threads
- Either activated immediately on construction, or subsequently
  - Via ::deferred and and initialize()
- A task scheduler init affects all subsequently created schedulers
  - Also wrt floating point settings
- Warning:
  - ensure that the task_scheduler_init is not automatically destroyed right after it has been created.

Explicitly managing the thread number

- Since TBB 4.0+, the task_scheduler_init has changed behaviour.
  - It will provide at least the specified thread number if enough HW resources are available
  - they may possibly be more than required
  - still the task scheduler affects _all_ TBB parallelism
- Other mechanism to control parallelism
- Task Groups
  - High level interface to the scheduler, allowing to set up a task repository served by computing threads
- Task arena
  - Intermediate interface toward the task scheduler
  - Can set up a limited number of thread to be used
    - As specified, or less if there are not enough resources available

Task Arena

```
task_arena(int max_concurrency = automatic,
           unsigned reserved_for_masters = 1)
```

- Create a task arena with limited concurrency, possibly reserving some threads to application threads
  - Can only reserve 0 or 1 threads up to TBB 4.4U1
  - TBB worker threads are needed to serve additional tasks enqueued to the arena

Task Arena example (Intel)

```
tbb::task_scheduler_init def_init;
// Use the default number of threads.
tbb::task_arena limited(2);
// No more than 2 threads in this arena.
tbb::task_group tg;
limited.execute([&]{
// Use at most 2 threads for this job.
    tg.run([]{ // run in task group
        tbb::parallel_for(1, N, unscalable_work());
    });
});
// Run another job concurrently with the loop above.
// It can use up to the default number of threads.
tbb::parallel_for(1, M, scalable_work());
// Wait for completion of the task group in the limited
arena.

limited.execute([&]{ tg.wait(); });
```