CHAPTER 9

The Pillars of Composability

In this chapter, we discuss composability: what it is, what characteristics make Threading Building Blocks (TBB) a composable threading library, and how to use a library like TBB to create scalable applications. C++ is a composable language, and TBB adds parallelism in a way that maintains composability. Composability with TBB is highly valuable because it means we are free to expose opportunities for parallelism without worrying about overloading the system. If we do not expose parallelism, we limit scaling.

Ultimately, when we say that TBB is a composable parallel library, we mean that developers can mix and match code that uses TBB freely anywhere they want. These uses of TBB can be serial, one after the other; they can be nested; they can be concurrent; they can be all within a single monolithic application; they can be spread across disjoint libraries; or they can be in different processes that execute concurrently.

It might not be obvious that parallel programming models have often had restrictions that were difficult to manage in complex applications. Imagine if we could not use "`while`" statements within an "`if`" statement, even indirectly in functions we call. Before TBB, equally difficult restrictions existed for some parallel programming models, such as OpenMP. Even the newer OpenCL standard lacks full composability.

The most frustrating aspect of non-composable parallel programming models is that there is such a thing as requesting too much parallelism. This is horrible, and something TBB avoids. In our experience, naïve users of non-composable models often overuse parallelism – and their programs crash from explosions in memory usage or they slow down to a crawl due to unbearable synchronization overheads. Concern about these issues can lead experienced programmers to expose too little parallelism, resulting in load imbalances and poor scaling. Using a composable programming model avoids the need to worry about this difficult balancing act.

© Intel Corporation 2019
M. Voss, R. Asenjo, J. Reinders, *Pro TBB*, https://doi.org/10.1007/978-1-4842-4398-5_9

Composability makes TBB extraordinarily reliable to use in both simple and complex applications. Composability is a design philosophy that allows us to create programs that are more scalable because we can expose parallelism without fear. In Chapter 1, we introduced the idea of the three-layer cake of parallelism that is common in many applications, reproduced here as Figure 9-1.

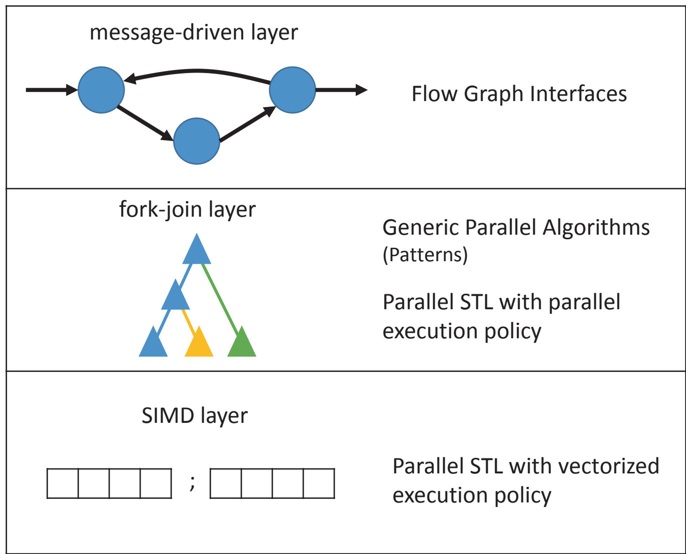


Figure 9-1. *The three layers of parallelism commonly found in applications and how they map to the high-level TBB parallel execution interfaces*

We covered the basics of the high-level interfaces shown in Figure 9-1 in the generic parallel algorithms in Chapter 2, the flow graph in Chapter 3, and Parallel STL in Chapter 4. Each of these high-level interfaces plays an important role in building up these layers of parallelism. And because they are all implemented using TBB tasks, and TBB is composable, we can safely combine them together to make complex, scalable applications.

What Is Composability?

Composability is, unfortunately, not a simple yes-or-no property of a programming model. Even though OpenMP has known composability issues for nested parallelism, it would be incorrect to label OpenMP as a non-composable programming model. If an application invokes OpenMP construct after OpenMP construct in series, this serial composition works just fine. It would likewise be an overstatement to say that TBB is a fully composable programming model that works well with all other parallel programming models in all situations. Composability is more accurately thought of as a measure of how well two programming models perform when composed in a specific way.

For example, let's consider two parallel programming models: model `A` and model `B`. Let's define T_A as the throughput of a kernel when it uses model `A` to express outer-level parallelism, and T_B as the throughput of the same kernel when it uses model B (without using model `A`) to express inner-level parallelism. If the programming models are composable, we would expect the throughput of the kernel using both outer and inner parallelism to be $T_{AB} >= \max(T_A, T_B)$, How much greater T_{AB} is than $\max(T_A, T_B)$ depends both on how efficiently the models compose with each other and on the physical properties of the targeted platform, such as the number of cores, the size of the memory, and so on.

Figure 9-2 shows the three general types of composition that we can use to combine software constructs: nested execution, concurrent execution, and serial execution. We say that TBB is a composable threading library because when a parallel algorithm using TBB is composed with other parallel algorithms in one of the three ways shown in Figure 9-2, the resulting code performs well, that is $T_{TBB+Other} >= \max(T_{TBB}, T_{Other})$.

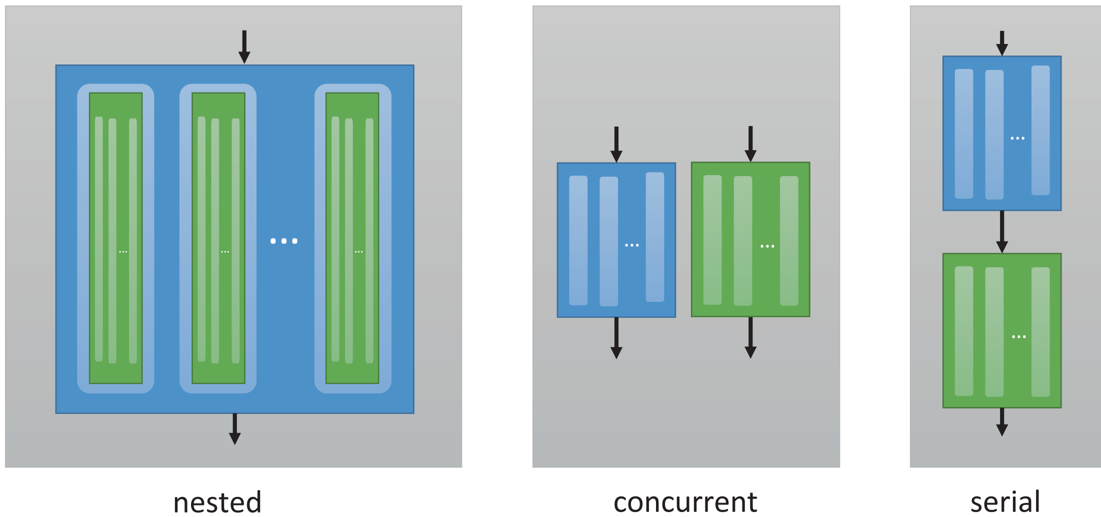


Figure 9-2. *The ways in which software constructs can be composed*

Before we discuss the features of TBB that lead to good composability, let's look at each composition type, the issues that can arise, and what performance impacts we can expect.

Nested Composition

In a *nested* composition, the machine executes one parallel algorithm inside of another parallel algorithm. The intention of a nested composition is almost always to add additional parallelism, and it can even exponentially increase the amount of work that can be executed in parallel as shown in Figure 9-3. Handling nested parallelism effectively was a primary goal in the design of TBB.

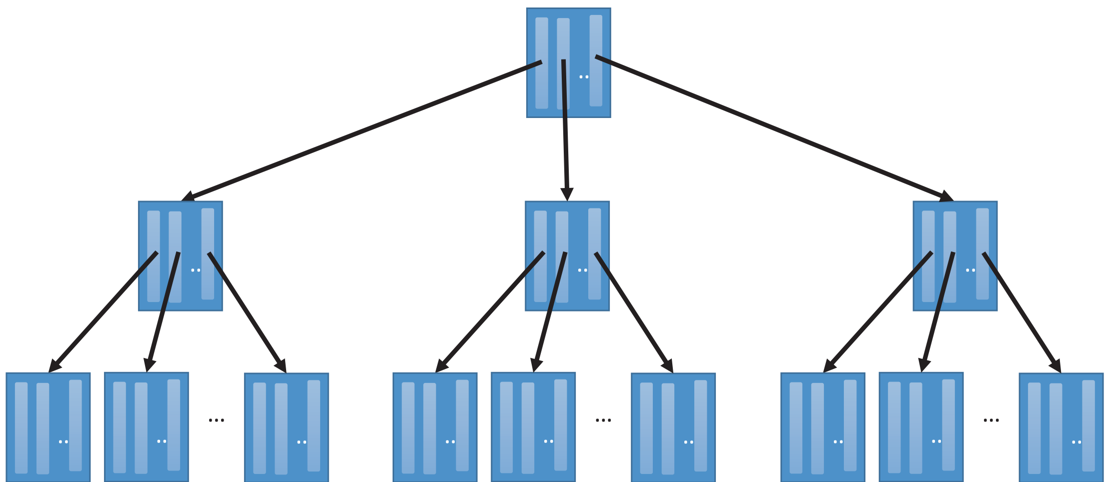


Figure 9-3. *Nested parallelism can lead to an exponential growth in the number of available parallel tasks (or when using a non-composable library, threads)*

In fact the algorithms provided by the TBB library in many cases depend on nested parallelism in order to create scalable parallelism. For example, in Chapter 2, we discussed how nested invocations of TBB's `parallel_invoke` can be used to create a scalable parallel version of quicksort. The Threading Building Blocks library is designed from the ground up to be an effective executor of nested parallelism.

In contrast to TBB, other parallel models may perform catastrophically bad in the presence of nested parallelism. A concrete example is the OpenMP API. OpenMP is a widely used programming model for shared-memory parallelism – and it is very effective for single level parallelism. However, it is a notoriously bad model for nested parallelism because mandatory parallelism is an integral part of its definition. In applications that have multiple levels of parallelism, each OpenMP parallel construct creates an additional team of threads. Each thread allocates stack space and also needs to be scheduled by the OS's thread scheduler. If the number of threads is very large, the application can run out of memory. If the number of threads exceeds the number of logical cores, the threads must share cores. Once the number of threads exceeds the number of cores, they tend to offer little benefit due to the oversubscription of the hardware resources, adding only overhead.

The most practical choice for nested parallelism with OpenMP is typically to turn off the nested parallelism completely. In fact, the OpenMP API provides an environment variable, `OMP_NESTED`, for the purpose of turning on or off nested parallelism. Because TBB has relaxed sequential semantics and uses tasks to express parallelism instead of

threads, it can flexibly adapt parallelism to the available hardware resources. We can safely leave nested parallelism on with TBB – there's no need for a mechanism to turn off parallelism in TBB!

Later in this chapter, we discuss the key features of TBB that make it very effective at executing nested parallelism, including its thread pool and work-stealing task scheduler. In Chapter 8, we examine nesting as a very important recurring theme (pattern) in parallel programming. In Chapter 12, we discuss features that allow us to influence the behavior of the TBB library when executing nested parallelism in order to create isolation and improve data locality.

Concurrent Composition

As shown in Figure 9-4, *concurrent* composition is when the execution of parallel algorithms overlap in time. Concurrent composition can be used to intentionally add additional parallelism, or it can arise by happenstance when two unrelated applications (or constructs in the same program) execute concurrently on the same system. Concurrent and parallel execution are not always the same thing! As shown in Figure 9-3, *concurrent execution* is when multiple constructs execute during the same time frame, while *parallel execution* is when multiple constructs execute simultaneously. This means that parallel execution is a form of concurrent execution but concurrent execution is not always parallel execution. Concurrent composition improves performance when it is effectively turned into parallel execution.

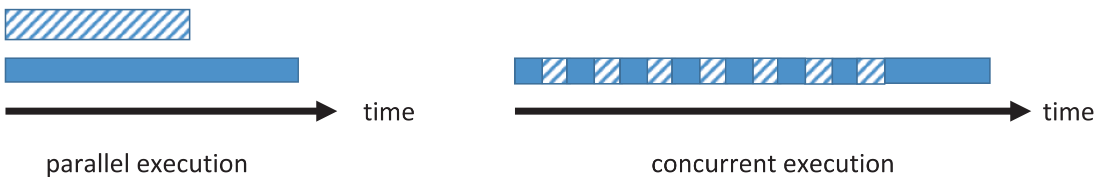


Figure 9-4. *Parallel vs. concurrent execution*

A concurrent composition of the two loops in Figure 9-5 is when a parallel implementation of loop 1 executes concurrently with a parallel implementation of loop 2, whether in two different processes or in two different threads in the same process.

```
// loop 1
for (int i = 0; i < N; ++i) {
  b[i] = f(a[i]);
}
```

```
// loop 2
for (int i = 0; i < M; ++i) {
  d[i] = g(c[i]);
}
```

Figure 9-5. *Two loops that execute concurrently*

When executing constructs concurrently, an arbitrator (a runtime library like TBB, the operating system or some combination of systems) is responsible for assigning system resources to the different constructs. If the two constructs require access to the same resources at the same time, then access to these resources must be interleaved.

Good performance for a concurrent composition might mean that the wall-clock execution time is as short as the time to execute the longest running construct, since all of the other constructs can execute in parallel with it (like in the parallel execution in Figure 9-4). Or, good performance might mean that the wall-clock execution time is no longer than the sum of the execution times of all the constructs if the executions need to be interleaved (like in the concurrent execution in Figure 9-4). But no system is ideal, and sources of both destructive and constructive interference make it unlikely that we get performance that exactly matches either of these cases.

First, there is the added cost of the arbitration. For example, if the arbitrator is the OS thread scheduler, then this would include the overheads of the scheduling algorithm; the overheads of preemptive multitasking, such as switching thread contexts; as well as the overheads of the OS's security and isolation mechanisms. If the arbitrator is a task scheduler in a user-level library like TBB, this cost is limited to the overheads of scheduling the tasks on to threads. If we express very fine-grained pieces of work, using many tasks scheduled on to a small set of threads has much lower scheduling overheads than using many threads directly, even though the tasks ultimately execute on top of threads.

Secondly, there is the performance impact from the concurrent use of shared system resources, such as the functional units, memory, and data caches. The overlapped execution of constructs can, for example, lead to changes in data cache performance – often an increase in cache misses but, in rare cases of constructive interference, possibly even a decrease in cache misses.

TBB's thread pool and its work-stealing task scheduler, discussed later in this chapter, help with concurrent composition as well, reduce arbitration overheads, and in many cases lead to task distributions that optimize resource usage. If TBB's default behaviors are not satisfactory, the features described in Chapters 11–14 can be used to mitigate negative impacts of resource sharing as needed.

Serial Composition

The final way to compose two constructs is to execute them serially, one after the other without overlapping them in time. This may seem like a trivial kind of composition with no implications on performance, but (unfortunately) it is not. When we use serial composition, we typically expect good performance to mean that there is no interference between the two constructs.

For example, if we consider the loops in Figure 9-6, the serial composition is to execute loop 3 followed by loop 4. We might expect that the time to complete each parallel construct when executed in series is no different than the time to execute that same construct alone. If the time it takes to execute loop 3 alone after parallelism is added using a parallel programming model A is $t_{3,A}$ and the time to execute loop 4 alone using a parallel programming model B is $t_{4,B}$, then we would expect the total time for executing the constructs in series is no more than the sum of the times of each construct, $t_{3,A} + t_{4,B}$.

```
// loop 3
for (int i = 0; i < N; ++i) {
  b[i] = f(a[i]);
}

// loop 4
for (int i = 0; i < N; ++i) {
  c[i] = f(b[i]);
}
```

Figure 9-6. *Two loops that are executed one after the other*

However, as with concurrent composition, there are sources of both destructive and constructive interference that can arise and cause actual execution times to diverge from this simple expectation.

In serial composition, the application must transition from one parallel construct to the next. Figure 9-7 shows ideal and nonideal transitions between constructs when using the same or different parallel programming models. In both ideal cases, there is no overhead, and we move immediately from one construct to the next. In practice, there is often some time required to clean up resources after executing a construct in parallel as well as some time required to prepare resources before the execution of the next construct.

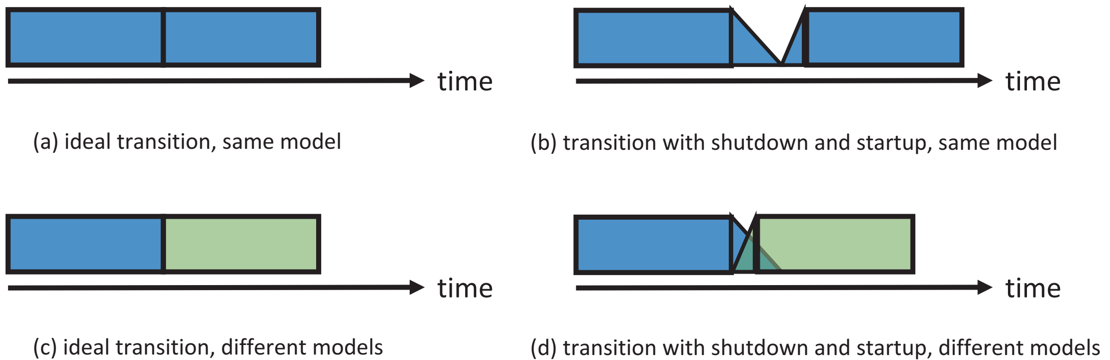


Figure 9-7. *Transitioning between the executions of different constructs*

When the same model is used, as shown in Figure 9-7(b), a runtime library may do work to shut down the parallel runtime only to have to immediately start it back up again. In Figure 9-7(d), we see that if two different models are used for the constructs, they may be unaware of each other, and so the shut-down of the first construct and the start-up, and even execution, of the next construct can overlap, perhaps degrading performance. Both of these cases can be optimized for – and TBB is designed with these transitions in mind.

And as with any composition, performance can be impacted by the sharing resources between the two constructs. Unlike with the nested or concurrent compositions, the constructs do not share resources simultaneously or in an interleaved fashion, but still, the ending state of the resources after one construct finishes can affect the performance of the next construct. For example, in Figure 9-6, we can see that loop 3 writes to array `b` and then loop 4 reads from array `b`. Assigning the same iterations in loop 3 and 4 to the same cores might increase data locality resulting in fewer cache misses. In contrast, an assignment of the same iterations to different cores can result in unnecessary cache misses.

The Features That Make TBB a Composable Library

The Threading Building Blocks (TBB) library is a composable library by design. When it was first introduced over 10 years ago, there was a recognition that as a parallel programming library targeted at all developers – not just developers of flat, monolithic applications – it had to address the challenges of composability head-on. The applications that TBB is used in are often modular and make use of third-party libraries

that may, themselves, contain parallelism. These other parallel algorithms may be intentionally, or unintentionally, composed with algorithms that use the TBB library. In addition, applications are typically executed in multiprogrammed environments, such as on shared servers or on personal laptops, where multiple processes execute concurrently. To be an effective parallel programming library for all developers, TBB has to get composability right. And it does!

While it is not necessary to have a detailed understanding of the design of TBB in order to create scalable parallel applications using its features, we cover some details in this section for interested readers. If you are happy enough to trust that TBB does the right thing and are not too interested in how, then you can safely skip the rest of this section. But if not, read on to learn more about why TBB is so effective at composability.

The TBB Thread Pool (the Market) and Task Arenas

The two features of the Threading Building Blocks library that are primarily responsible for its composability are its *global thread pool (the market)* and *task arenas*. Figure 9-8 shows how the global thread pool and a single default task arena interact in an application that has a single main thread; for simplicity, we will assume that there are `P=4` logical cores on the target system. Figure 9-8(a) shows that the application has `1` application thread (the main thread) and a global thread pool of workers that is initialized with `P-1` threads. The workers in the global thread pool execute dispatchers (represented by the solid boxes). Initially, each thread in the global thread pool sleeps while waiting for an opportunity to participate in parallel work. Figure 9-8(a) also shows that a single default task arena is created. Each application thread that uses TBB is given its own task arena to isolate its work from the work of the other application threads. In Figure 9-8(a), there is only a single task arena, since there is only a single application thread. When the application thread executes a TBB parallel algorithm, it executes a dispatcher tied to that task arena until the algorithm is complete. While waiting for the algorithm to complete, the master thread can participate in executing tasks that are spawned into the arena. The main thread is shown filling the slot reserved for a master thread.

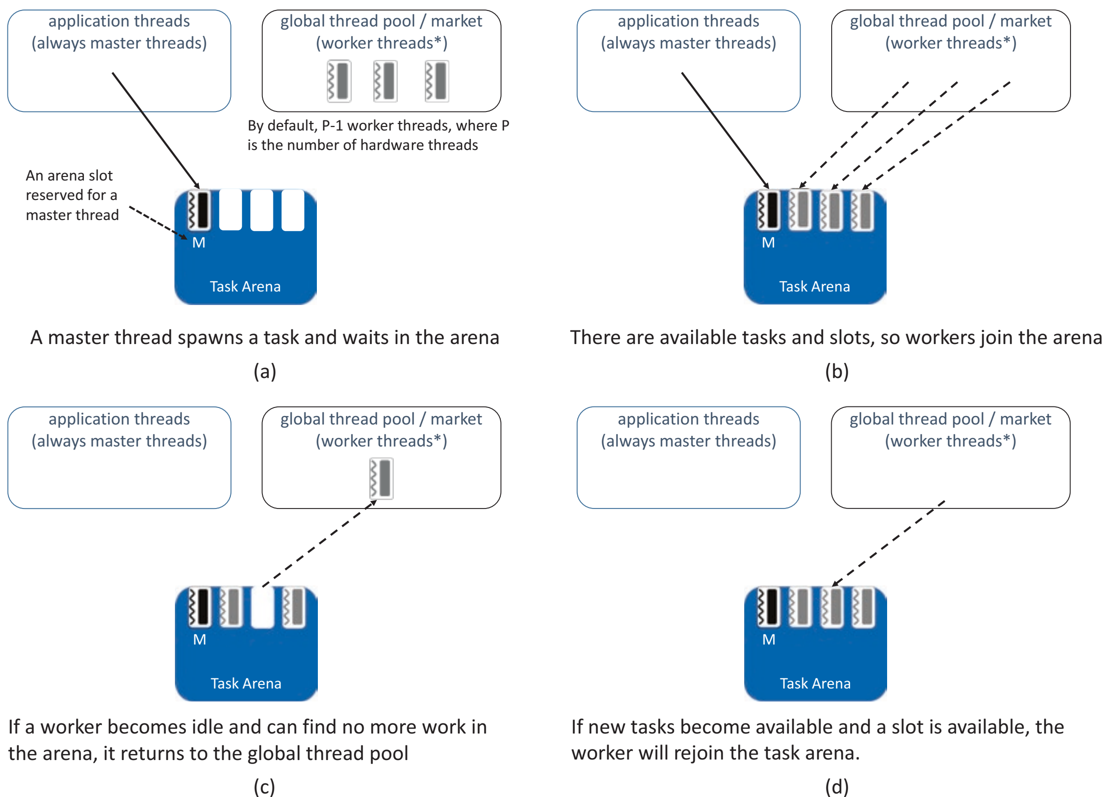


***Figure 9-8.** In many applications, there is a single main thread, and the TBB library, by default, creates P-1 worker threads to participate in the execution of parallel algorithms*

When a master thread joins an arena and first spawns a task, the worker threads sleeping in the global thread pool wake up and migrate to the task arena as shown in Figure 9-8(b). When a thread joins a task arena, by filling one of its slots, its dispatcher can participate in executing tasks that are spawned by other threads in that arena, as well as spawn tasks that can be seen and stolen by the other threads' dispatchers that are connected to the arena. In Figure 9-8, there are just enough threads to fill the slots in the task arena, since the global thread pool creates `P-1` threads and the default task arena has enough slots for `P-1` threads. Typically, this is exactly the number of threads we want, since the main thread plus `P-1` worker threads will fully occupy the cores in the machine without oversubscribing them. Once the task arena is fully occupied, the spawning of tasks does not wake up additional threads waiting in the global thread pool.

Figure 9-8(c) shows that when a worker thread becomes idle and can find no more work to do in its current task arena, it returns to the global thread pool. At that point,

the worker could join a different task arena that needs workers if one is available, but in Figure 9-8, there is only a single task arena, so the thread will go back to sleep. If later more tasks become available, the threads that have returned to the global thread pool will wake back up and rejoin the task arena to assist with the additional work as shown in Figure 9-8(d).

The scenario outlined in Figure 9-8 represents the very common case of an application that has a single main thread and no additional application threads, and where no advanced features of TBB are used to change any defaults. In Chapters 11 and 12, we will discuss advanced TBB features that will allow us to create more complicated examples like the one shown in Figure 9-9. In this more complicated scenario, there are many application threads and several task arenas. When there are more task arena slots than worker threads, as is the case in Figure 9-8, the worker threads are divided in proportion to the need of each task arena. So, for example, a task arena with twice as many open slots as another task arena will receive roughly twice as many worker threads.

Figure 9-9 highlights a few other interesting points about task arenas. By default, there is one slot reserved for a master thread, like in Figure 9-8. However as shown by the right two task arenas in Figure 9-9, a task arena can be created (using advanced features that we discuss in later chapters) that reserves multiple slots for master threads or no slots at all for master threads. A master thread can fill any slot, while threads that migrate to an arena from the global thread pool cannot fill slots reserved for masters.

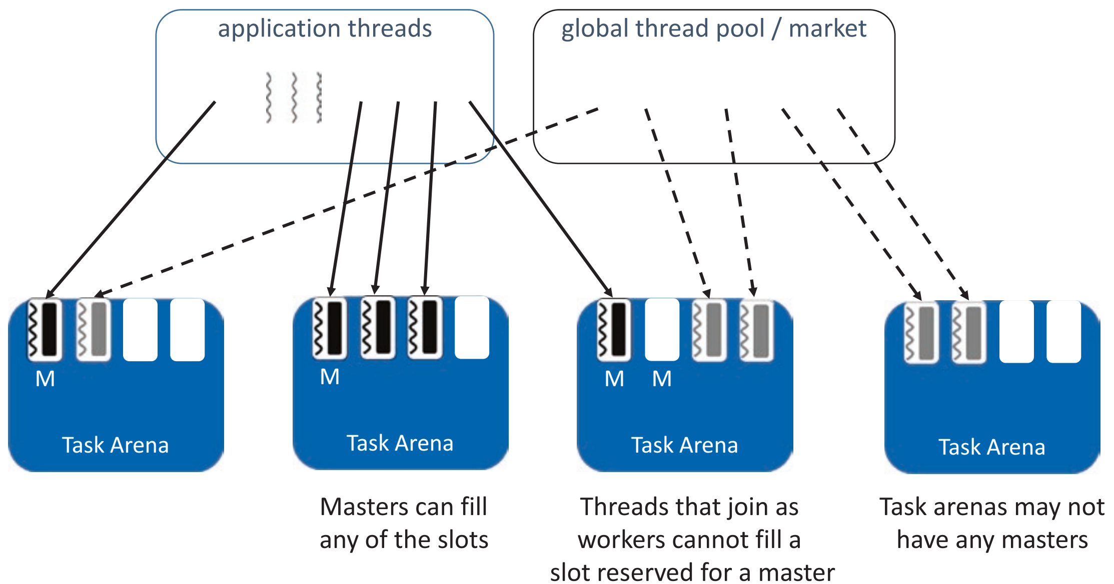


Figure 9-9. *A more complicated application with many native threads and task arenas*

Regardless of how complicated our application though, there is always a single global thread pool. When the TBB library is initialized, it allocates threads to the global thread pool. In Chapter 11, we discuss features that allow us to change the number of threads that are allocated to the global thread pool at initialization, or even dynamically, if we need to. But this limited set of worker threads is one reason that TBB is composable, since it prevents unintended oversubscription of the platform's cores.

Each application thread also gets its own implicit task arena. A thread cannot steal a task from a thread that is in another task arena, so this nicely isolates the work done by different application threads by default. In Chapter 12, we will discuss how application threads can choose to join other arenas if they want to – but by default they have their own.

The design of TBB makes applications and algorithms that use TBB tasks compose well when executed nested, concurrently, or serially. When nested, TBB tasks generated at all levels are executed within the same arena using only the limited set of worker threads assigned to the arena by the TBB library, preventing an exponential explosion in the number of threads. When run concurrently by different master threads, the worker threads are split between the arenas. And when executed serially, the worker threads are reused across the constructs.

Although the TBB library is not directly aware of the choices being made by other parallel threading models, the limited number threads it allocates in its global thread pool also limits its burden on those other models. We will discuss this in more detail later in this chapter.

The TBB Task Dispatcher: Work Stealing and More

The Threading Building Blocks scheduling strategy is often described as *work stealing*. And this is mostly true. Work stealing is a strategy that is designed to work well in dynamic environments and applications, where tasks are spawned dynamically and execution occurs on a multiprogrammed system. When work is distributed by work stealing, worker threads actively look for new work when they become idle, instead of having work passively assigned to them. This pay-as-you-go approach to work distribution is efficient because it does not force threads to stop doing useful work just so they can distribute part of their work to other idle threads. Work stealing moves these overheads on to the idle threads – which have nothing better to do anyway! Work-stealing schedulers stand in contrast to *work-sharing* schedulers, which assign tasks to worker threads up front when tasks are first spawned. In a dynamic environment, where

tasks are spawned dynamically and some hardware threads may be more loaded than others, work-stealing schedulers are more reactive, resulting in better load balancing and higher performance.

In a TBB application, a thread participates in executing TBB tasks by executing a task dispatcher that is attached to a specific task arena. Figure 9-10 shows some of the important data structures that are maintained in each task arena and each per-thread task dispatcher.

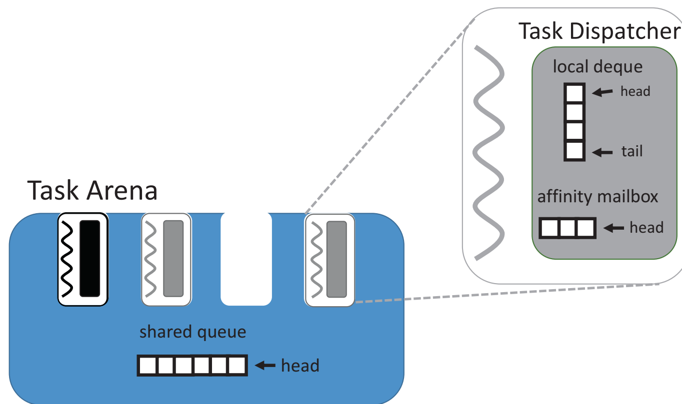


Figure 9-10. *The queues in a task arena and in the per-thread task dispatchers*

For the moment, let us ignore the shared queue in the task arena and the affinity mailbox in the task dispatcher and focus only on the local deque[1] in the task dispatcher. It is the local deque that is used to implement the work-stealing scheduling strategy in TBB. The other data structures are used to implement extensions to work stealing, and we will come back to those later.

In Chapter 2, we discussed the different kinds of loops that are implemented by the generic parallel algorithms included in the TBB library. Many of them depend on the concept of a Range, a recursively divisible set of values that represent the iteration space of the loop. These algorithms recursively divide a loop's Range, using *split tasks* to divide the Range, until they reach a good size to pair with the loop body to execute as a *body task*. Figure 9-11 shows an example distribution of tasks that implement a loop pattern. The top-level task t_0 represents the splitting of the complete Range, which is recursively split down to the leaves where the loop body is applied to each given subrange. With

[1]Deque means *double ended queue*, a data structure, not to be confused with dequeue, which is the action of removing items from a queue.

the distribution shown in Figure 9-11, each thread executes body tasks that execute across a contiguous set of iterations. Since nearby iterations often access nearby data, this distribution tends to optimize for locality. And because threads execute tasks within isolated task trees, once a thread gets an initial subrange to work on, it can execute on that tree without interacting much with the other threads.

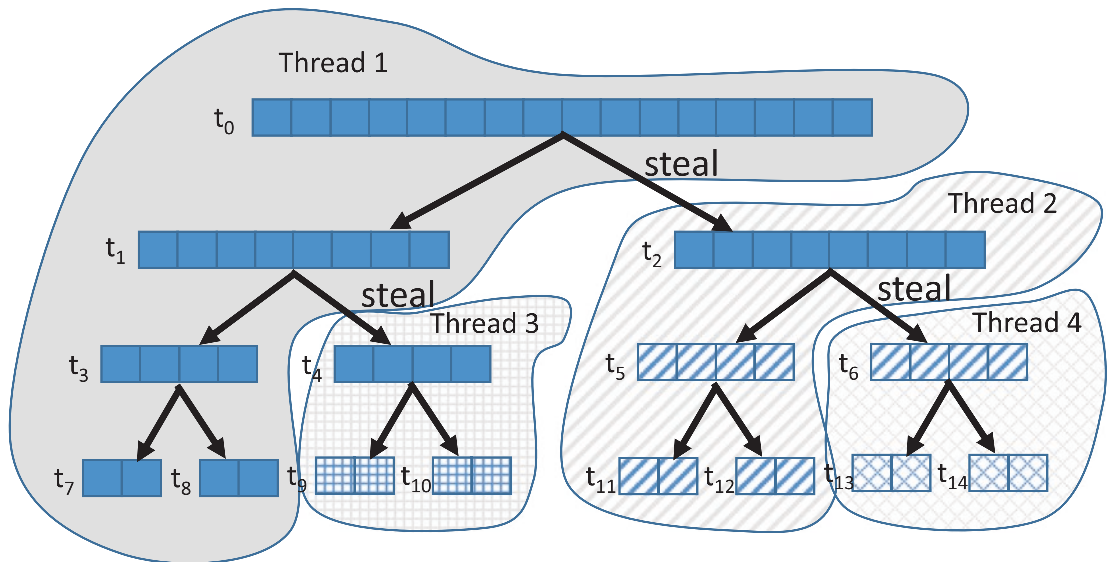


Figure 9-11. *A distribution of tasks that implements a loop pattern*

The TBB loop algorithms are examples of *cache-oblivious* algorithms. Perhaps, ironically, cache-oblivious algorithms are designed to highly optimize the use of CPU data caches – they just do this without knowing the details about cache or cache line sizes. As with the TBB loop algorithms, these algorithms are typically implemented using a divide-and-conquer approach that recursively divides data sets into smaller and smaller pieces that eventually fit into the data caches regardless of their sizes. We cover cache-oblivious algorithms in more detail in Chapter 16.

The TBB library task dispatchers use their local deques to implement a scheduling strategy that is optimized to work with cache-oblivious algorithms and create distributions like the one in Figure 9-11. This strategy is sometimes called a depth-first work, breadth-first steal policy. Whenever a thread *spawns* a new task – that is, makes it available to its task arena for execution – that task is placed at the head of its task dispatcher's local deque. Later, when it finishes the task it is currently working on and needs a new task to execute, it attempts to take work from the head of its local deque,

taking the task it most recently spawned as shown in Figure 9-12. If, however, there is no task available in a task dispatcher's local deque, it looks for nonlocal work by randomly selecting another worker thread in its task arena. We call the selected thread a *victim* since the dispatcher is planning to steal a task from it. If the victim's local deque is not empty, the dispatcher takes a task from the tail of the victim thread's local deque, as shown in Figure 9-12, taking the task that was least recently spawned by that thread.

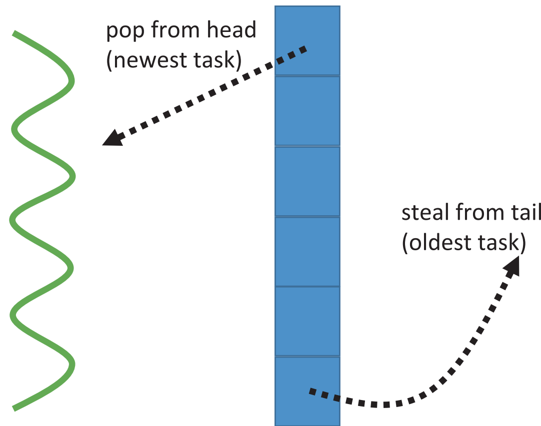
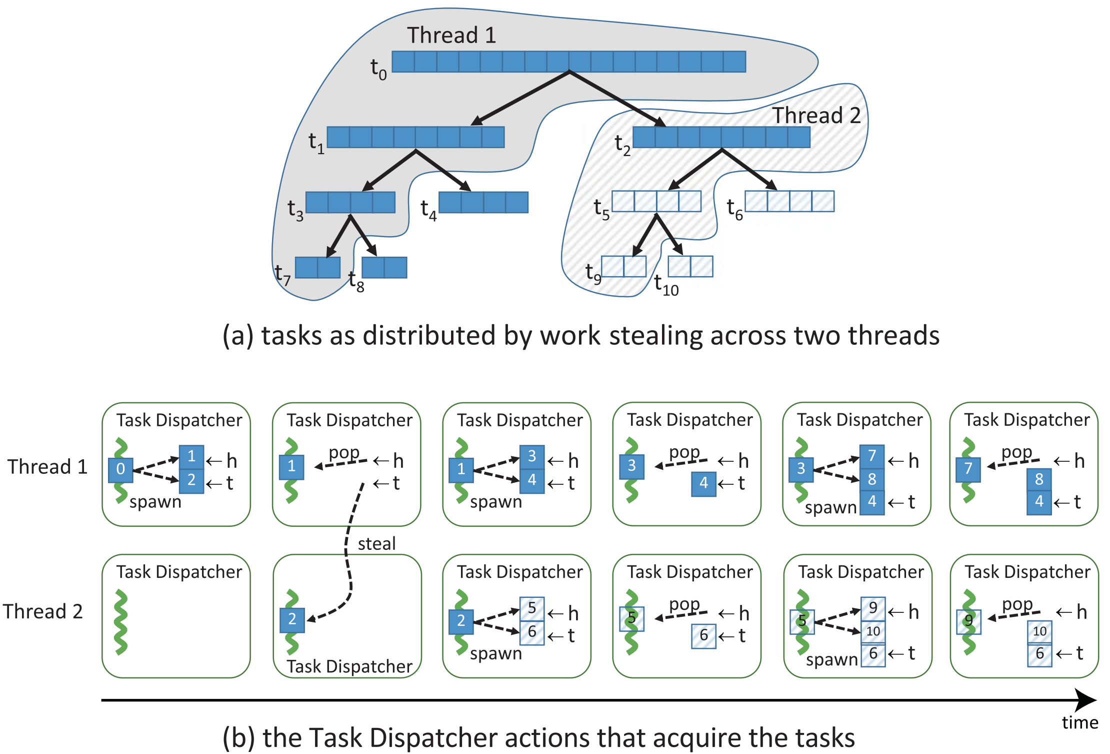


Figure 9-12. *The policy used by the task dispatcher to take local tasks from the head of the local deque but steal tasks from the tail of a victim thread's deque*

Figure 9-13 shows a snapshot of how tasks may be distributed by the TBB scheduling policy when executed using only two threads. The tasks shown in Figure 9-13 are a simplified approximation of a TBB loop algorithm. The TBB algorithm implementations are highly optimized and so may divide some tasks recursively without spawning tasks or use techniques like scheduler bypass (as described in Chapter 10). The example shown in Figure 9-13 assumes that each split and body task is spawned into the task arena – this is not really the case for the optimized TBB algorithms; however, this assumption is useful for illustrative purposes here.



(a) tasks as distributed by work stealing across two threads

(b) the Task Dispatcher actions that acquire the tasks

Figure 9-13. *A snapshot of how tasks may be distributed across two threads and the actions the two task dispatchers took to acquire the tasks. Note: the actual implementations of TBB loop patterns use scheduler bypass and other optimizations that remove some spawns. Even so, the stealing and execution order will be similar to this figure.*

In Figure 9-13, Thread 1 starts with the root task and initially splits the Range into two large pieces. It then goes depth-first down one side of the task tree, splitting tasks until it reaches the leaf where it applies the body to a final subrange. Thread 2, which is initially idle, steals from the tail of Thread 1's local deque, providing itself with the second large piece that Thread 1 created from the original Range. Figure 9-13(a) is a snapshot in time, for example tasks t_4 and t_6 have not yet been taken by any thread. If two more worker threads are available, we can easily imagine that we get the distribution shown in Figure 9-11. At the end of the timeline in Figure 9-13(b), Thread 1 and 2 still have tasks in their local deques. When they pop the next task, they will grab the leaves that are contiguous with the tasks they just completed.

We shouldn't forget when looking at Figures 9-11 and 9-13 that the distribution shown is only one possibility. If the work per iteration is uniform and none of the cores are oversubscribed, we will likely get the equal distributions shown. Work stealing, however, means that if one of the threads is executing on an overloaded core, it will steal less often and subsequently acquire less work. The other threads will then pick up the slack. A programming model that only provides a static, equal division of iterations to cores would be unable to adapt to such a situation.

As we noted earlier, the TBB task dispatchers however are not just work-stealing schedulers. Figure 9-14 provides a simplified pseudo-code representation of the entire task dispatch loop. We can see lines commented as "execute the task," "take a task spawned by this thread," and "steal a task." These points implement the work-stealing strategy that we just outlined here, but we can see that there are other actions interleaved in the task dispatch loop as well.

The line labeled "scheduler bypass" implements an optimization used to avoid task scheduling overheads. If a task knows exactly which task the calling thread should execute next, it can directly return it, avoiding some of the overheads of task scheduling. As users of TBB, this is likely something we will not need to use directly, but you can learn more about in Chapter 10. The highly optimized TBB algorithms and flow graph do not use straightforward implementations like that shown in Figure 9-13 but instead rely on optimizations, like scheduler bypass, to provide best performance.

The line labeled "take a task with affinity for this thread" looks into the task dispatcher's affinity mailbox to find a task before it attempts to steal work from a random victim. This feature is used to implement task-to-thread affinity, which we describe in detail in Chapter 13.

And the line labeled "take a task from the arena's shared queue" in Figure 9-14 is used to support enqueued tasks – tasks that are submitted to the task arena outside of the usual spawning mechanism. These enqueued tasks are used for work that needs to be scheduled in a roughly first-in-first out order or for fire-and-forget tasks that need to be eventually executed but are not part of a structured algorithm. Task enqueuing will be covered in more detail in Chapter 10.

```
if this thread is a master
  t_next = the first task
else if this thread is a worker
  t_next = steal of task from a random thread's local deque
endif
do
  do
    do
      do
        do
          do
            // execute the task
            t = t_next
            t_next = t->execute()
          while t_next is a valid task // scheduler bypass
          if I'm a master and my algorithm is done
            return
          else if this was the last child of a parent task
            t_next = t->parent
          end if
        while t_next is a valid task
        // take a task spawned by this thread
        t_next = pop from end of local deque
      while t_next is a valid task
      // take a task with affinity for this thread
      t_next = get task from affinity mailbox
    while t_next is a valid task
    // take a task form the arena's shared queue
    t_next = pop from front (approximately) of shared queue
  while t_next t is a valid task
  // steal a task
  t_next = steal task from front of random thread's local deque
while t_next is a valid task

// I'm a worker and there's nothing left to do:
return myself to the global thread pool
```

Figure 9-14. *Pseudo-code for an approximation of the TBB task dispatch loop*

The TBB dispatcher shown in Figure 9-14 is a user-level, nonpreemptive task scheduler. An OS thread scheduler is much more complex, since it will need to deal with not only a scheduling algorithm but also thread preemption, thread migration, isolation, and security.

Putting It All Together

The previous sections describe the design that allows TBB algorithms and tasks to execute efficiently when composed in various ways. Earlier, we also claimed that TBB performs well when mixed with other parallel models too. With our newly acquired knowledge, let's revisit our composition types to convince ourselves that TBB is in fact a composable model because of its design.

In this discussion, we will compare against a hypothetical non-composable thread library, the Non-Composable Runtime (NCR). Our fictional NCR includes parallel constructs that require mandatory parallelism. Each NCR construct will require a team of P threads, which need to be exclusively used for the construct until it is finished – they cannot be shared by other concurrently executing or nested NCR constructs. NCR will also create its threads at the first use of a NCR construct but will not put its threads to sleep after its constructs end – it will keep them actively spinning, using up CPU cycles, so that they can respond as quickly as possible if another NCR construct is encountered. Behaviors like these are not uncommon in other parallel programming models. OpenMP parallel regions do have mandatory parallelism, which can lead to big trouble when the environment variable OMP_NESTED is set to "true." The Intel OpenMP runtime library also provides the option to keep the worker threads actively spinning between regions by setting the environment variable `OMP_WAIT_POLICY` to "active." To be fair, we should make it clear that the Intel OpenMP runtime defaults are `OMP_NESTED=false` and `OMP_WAIT_POLICY=passive`, so these non-composable behaviors are not the default. But as a point of comparison, we use NCR as a strawman to represent a very badly behaved, non-composable model.

Now, let's look out how well TBB composes with itself and with NCR. As a proxy for performance, we will look at oversubscription since the more oversubscribed a system is, the more scheduling and destructive sharing overheads it will likely incur. Figure 9-15 shows how our two models nest with themselves. When a TBB algorithm is nested inside of a TBB algorithm, all of the generated tasks will execute in the same arena and share the `P` threads. However, NCR shows an explosion in threads since each nested construct will need to assemble its own team of `P` threads, ultimately needing P^2 threads for even a two-level deep nesting.

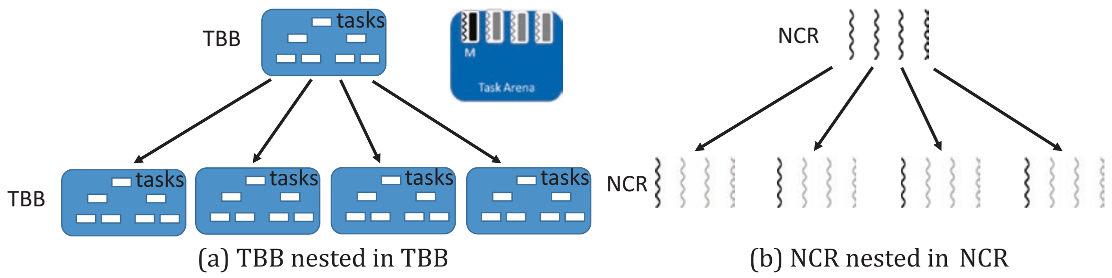


Figure 9-15. *The number of threads used for TBB nested in TBB and for a non-composable runtime (NCR) nested in NCR*

Figure 9-16 shows what happens when we combine the models. It doesn't matter how many threads execute TBB algorithms concurrently – the number of TBB worker threads will remain capped at `P-1`! When TBB is nested inside of NCR, we therefore use at most only `2P-1` threads: `P` threads from NCR, which will act like master threads in the nested TBB algorithms, and the `P-1` TBB worker threads. If a NCR construct is nested inside of TBB however, each TBB task that executes a NCR construct will need to assemble a team of P threads. One of the threads might be the thread executing the outer TBB task, but the other `P-1` threads will need to be created by or obtained from the NCR library. We therefore wind up with the `P` threads from TBB each executing in parallel and each using an additional `P-1` threads, for a total of P^2 threads. We can see from Figures 9-15 and 9-16 that when TBB is nested inside of even a badly performed model, it behaves well – unlike a non-composable model like NCR.

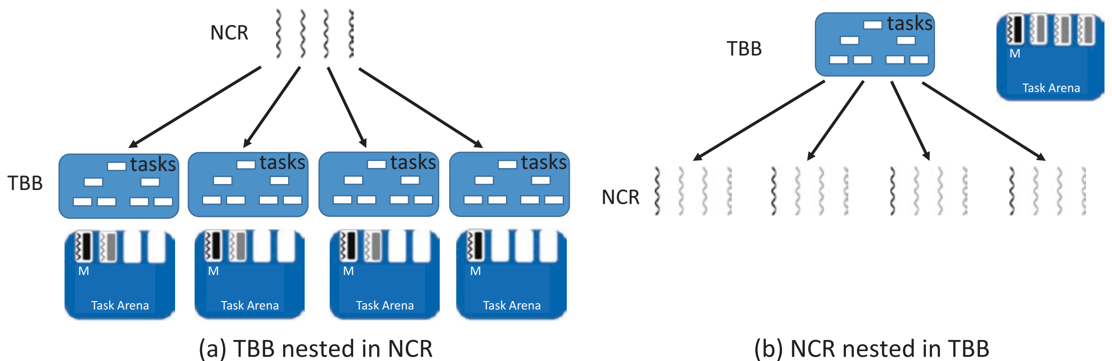


Figure 9-16. *When TBB and a non-composable runtime (NCR) are nested inside of each other*

When we look at concurrent execution, we need to consider both single-process concurrent, when parallel algorithms are executed by different threads in the same process concurrently, and multiprocess concurrency. The TBB library has single global thread pool per process – but does not share the thread pool across processes. Figure 9-17 shows the number of threads used for different combinations of concurrent executions for the single-process case. When TBB executes concurrently with itself in two threads, each thread gets its own implicit task arena, but these arenas share the P-1 worker threads; the total number of threads therefore is P+1. NCR uses a team of P threads per construct, so it uses 2P threads. And likewise, since TBB and NCR do not share thread pools, they will use 2P threads when executing concurrently in a single process.

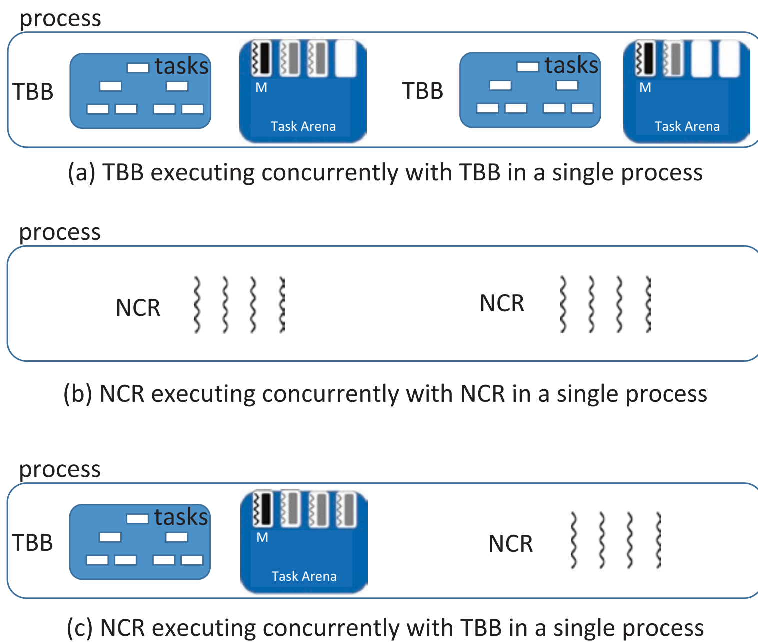


Figure 9-17. *The number of threads used for concurrent executions of TBB algorithms and non-composable runtime (NCR) constructs in a single process*

Figure 9-18 shows the number of threads used for different combinations of concurrent executions for the multiprocess case. Since TBB creates a global thread pool per-process, it no longer has an advantage in this case over NCR. In all three cases, 2P threads are used.

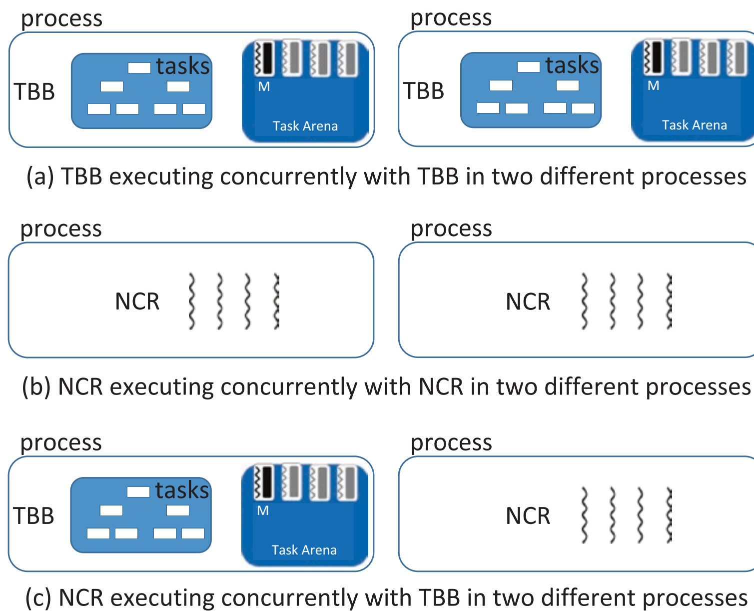


Figure 9-18. *The number of threads used for concurrent executions of TBB constructs and NCR constructs in two different processes*

Finally, let's consider the serial composition case, when one algorithm or construct is executed after another. Both TBB and NCR will compose well serially with other uses of their own libraries. If the delay is short, the TBB threads will still be in the task arena, since they actively search for work for a very short time once they run out of work. If the delay is long between TBB algorithms, the TBB worker threads will return to the global thread pool and migrate back to the task arena when new work becomes available. The overhead for this migration is very small, but non-negligible. Even so, typically the negative impact will be very low. Our hypothetical non-composable runtime (NCR) never sleeps, so it will always be ready to execute the next construct – no matter how long the delay. From a composability perspective, the more interesting cases are when we combine NCR and TBB together as shown in Figure 9-17. TBB quickly puts its threads to sleep after an algorithm ends, so it will not negatively impact an NCR construct that follows it. In contrast, the exceptionally responsive NCR library will keep it threads active, so a TBB algorithm that follows an NCR construct will be forced to fight these spinning threads for processor resources. TBB is clearly the better citizen because its design accounts for serial composability with other parallel models.

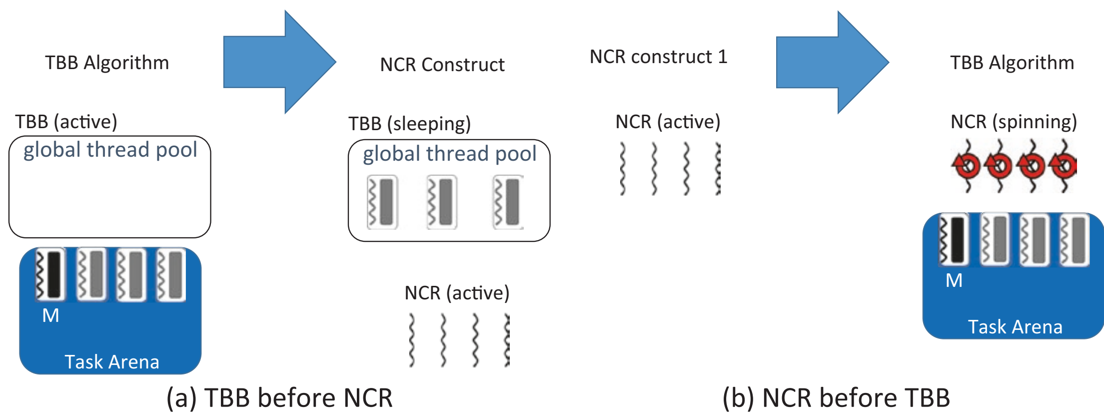


Figure 9-19. *The number of threads used for consecutive executions of TBB constructs and constructs that use mandatory parallelism*

Figures 9-15 through 9-19 demonstrate that TBB composes well with itself and its negative impact on other parallel models is limited due to its composable design. TBB algorithms efficiently compose with other TBB algorithms – but also are good citizens in general.

Looking Forward

In later chapters, we cover a number of topics that expand on themes raised in this chapter.

Controlling the Number of Threads

In Chapter 11, we describe how to use the `task_scheduler_init`, `task_arena`, and `global_control` classes to change the number of threads in the global thread pool and control the number of slots allocated to task arenas. Often, the defaults used by TBB are the right choice, but we can change these defaults if needed.

Work Isolation

In this chapter, we saw that each application thread gets its own implicit task arena by default to isolate its work from the work of other application threads. In Chapter 12, we discuss the function `this_task_arena::isolate`, which can be used in the uncommon

situations when work isolation is necessary for correctness. We will also discuss class `task_arena`, which is used to create explicit task arenas that can be used to isolate work for performance reasons.

Task-to-Thread and Thread-to-Core Affinity

In Figure 9-10, we saw that each task dispatcher not only has a local deque but also has an affinity mailbox. We also saw in Figure 9-14 that when a thread has no work left in its local deque, it checks this affinity mailbox before it attempts random work stealing. In Chapter 13, we discuss ways to create task-to-thread affinity and thread-to-core-affinity by using the low-level features exposed by TBB tasks. In Chapter 16, we discuss features like Ranges and Partitioners that are used by the high-level TBB algorithms to exploit data locality.

Task Priorities

In Chapter 14, we discuss task priorities. By default, the TBB task dispatchers view all tasks as equally important and simply try to execute tasks as quickly as possible, favoring no specific tasks over others. However, the TBB library does allow developers to assign low, medium, and high priorities to tasks. In Chapter 14, we will discuss how to use these priorities and their implications on scheduling.

Summary

In this chapter, we stressed the importance of composability and highlighted that we get it automatically if we use TBB as our parallel programming model. We started this chapter by discussing the different ways in which parallel constructs might be composed with each other and the issues that stem from each type of composition. We then described the design of the TBB library and how this design leads to composable parallelism. We concluded by revisiting the different composition types and compared TBB to a hypothetical non-composable runtime (NCR). We saw that TBB composes well with itself but also is a good citizen when combined with other parallel models.

For More Information

Cilk is a parallel model and platform that was one of key inspirations for the initial TBB scheduler. It provides a space efficient implementation of a work-stealing scheduler as described in

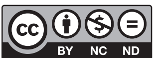
> Robert D. Blumofe and Charles E. Leiserson. 1993. Space-efficient scheduling of multithreaded computations. In Proceedings of the twenty-fifth annual ACM symposium on Theory of computing (STOC '93). ACM, New York, NY, USA, 362–371.

TBB provides generic algorithms implemented using tasks that execute on top of threads. By using TBB, developers can use these high-level algorithms instead of using low-level threads directly. For a general discussion of why using threads directly as a programming model should be avoided, see

> Edward A. Lee, "The Problem with Threads." Computer, 39, 5 (May 2006), 33–42.

In some ways, we've used the OpenMP API as a strawman non-composable model in this chapter. OpenMP is in fact a very effective programming model that has a wide user base and is especially effective in HPC applications. You can learn more about OpenMP at

> www.openmp.org



Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.